LensTools Documentation

Release 0.5

Andrea Petri

February 12, 2016

Contents

This python package collects together a suite of widely used analysis tools in Weak Gravitational Lensing

CHAPTER 1

Summary

This python add-on will handle basically every operation you will need to perform on Weak Lensing survey data; the distribution includes a range of tools in image analysis, statistical processing and numerical theory predictions and supports multiprocessing using the mpi4py module. This package includes many useful features, including:

- Measure units handling through astropy
- Complete flexibility and easy customization of input/output formats
- Efficient measurements of power spectrum, PDF, Minkowski functionals and peak counts of convergence maps
- Survey masks
- Efficient E and B mode decompositions of shear maps
- Artificial noise generation engines
- All functionality of pandas DataFrame readily available (and improved!)
- Easy to compute parameter statistical inferences
- Easy input/output from N-body simulation snapshots in the Gadget2 binary format
- Interfaces with existing simulation sets
- Ray Tracing simulations
- CPU vectorization of expensive computations via numpy
- Easy multiprocessing and cluster deployment via the mpi4py module
- *Future prospect*: taking advantage of numpy offload capabilities to Intel Xeon Phi coprocessors to boost performance (planned)

CHAPTER 2

Installation

The easiest way is to install through pip

```
pip install lenstools
```

An alternative is to install from source by cloning or forking the github repository to download the source and build it manually. F First clone the repository (the original one, or your fork):

```
git clone https://github.com/apetri/LensTools
```

Then, inside the LensTools directory build the source:

```
python setup.py build
```

If you want to test the build before installing to your system, look to the instructions in Test. Once you are satisfied install the package (might require root privileges depending on the install location):

```
python setup.py install
```

CHAPTER 3

Dependencies

The core features require the standard numpy, scipy , and additionally astropy (mainly for the cosmology and measure units support) and emcee (from which LensTools borrows the MPI Pool utility), and the Test suite requires additionally the matplotlib package. matpoltlib should eventually be installed if you want to use the plotting engines of LensTools. If you want to run the calculations in parallel on a computer cluster you will need to install mpi4py (a python wrapper for the MPI library). Installation of all these packages is advised (if you run astrophysical data analyses you should use them anyway). One of the lenstools features, namely the `Design` class, requires that you have a working version of GSL to link to; if you don't have one, just hit *enter* during the installation process and the package will work correctly without this additional feature. The installation if the NICAEA bindings additionally requires a working installation of the fftw3 library.

CHAPTER 4

Test

To check that everything works before installing you can run the pre implemented test suite that comes with the source code. First you will need to install pytest, then you need to download some data files (mainly FITS images) that the test suite depends on. You need to set the environment variable LENSTOOLS_DATA to the path where you want your data to be downloaded (for a manual download the data file can be found here, it is almost 250MB). After that, in a python shell, type

```
import lenstools
lenstools.dataExtern()
```

That should make sure the data directory is downloaded and available and should return the full path of the data directory. After that operation has completed create a Test directory and run the tests:

```
mkdir Test
cd Test
py.test --pyargs lenstools.tests
```

Each test, if successful, will produce some output (PNG plots, text files, etc...)

CHAPTER 5

Weak Lensing simulations

5.1 The LensTools Weak Lensing Simulation pipeline

This document is a hands–on tutorial on how to deploy the lenstools weak lensing simulation pipeline on a computer cluster. This will enable you to run your own weak lensing simulations and produce simulated weak lensing fields (shear and convergence) starting from a set of cosmological parameters.

5.1.1 Pipeline workflow

A scheme of the pipeline workflow can be seen in the figure below. The starting point is to select a set of cosmological parameters that are supposed to describe your mock universe (i.e. a combination of $(H_0, \Omega_m, \Omega_b, \Omega_\Lambda, w_0, w_a, \sigma_8, n_s)$). The first step in the pipeline is running several independent N–body simulations that will give an accurate picture of the dark matter density field of the universe at different redshifts. Once the N–body boxes are produced, the multi–lens–plane algorithm is used to trace the deflection of light rays as they go through the simulated density boxes. This will allow to compute background galaxies shape distortions and/or weak lensing quantities such as the shear $\gamma_{1,2}$ and the convergence κ (and, with a limited accuracy, the rotation angle ω).

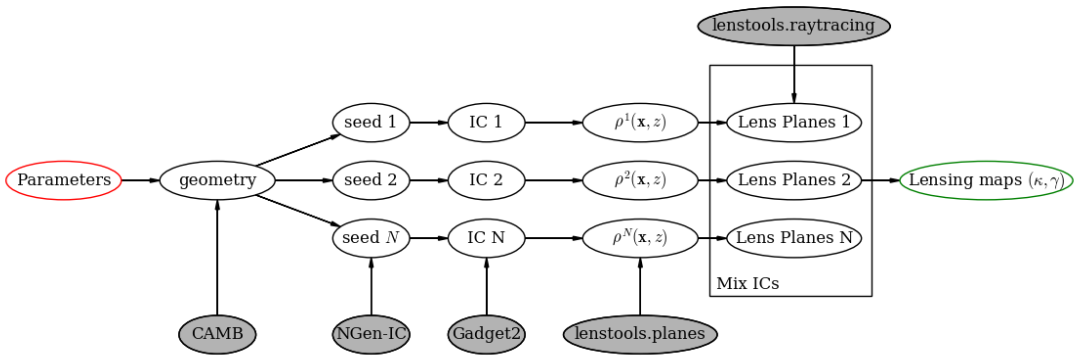


Fig. 5.1: A scheme of the pipeline workflow

In the remainder of the document a detailed scheme of the pipeline is illustrated.

5.1.2 The directory tree

lenstools provides a set of routines for managing the simulations directory tree, which is crucial for organizing the produced files in a sensible way. The first step in the process is creating a new batch of simulations. The way you accomplish this is through a *SimulationBatch* object.

```
>>> from lenstools.pipeline.simulation import SimulationBatch
>>> from lenstools.pipeline.settings import EnvironmentSettings
```

You will need to choose where you want to store your files: in each simulation batch there are two distinct locations files will be saved in. The "home" location is reserved for small files such as code parameter files, tabulated power spectra and other book–keeping necessary files. The "storage" location is used to store large production files, such as N–body simulation boxes, lensing planes and weak lensing maps. These locations need to be specified upon the batch creation

```
>>> environment = EnvironmentSettings(home="SimTest/Home",storage="SimTest/Storage")
>>> batch = SimulationBatch(environment)
```

or, if your settings are contained in a file called *environment.ini*, you can run the following

```
>>> batch = SimulationBatch.current("environment.ini")
```

The contents of *environment.ini* should look something like

```
[EnvironmentSettings]

home = SimTest/Home
storage = SimTest/Storage
```

You will need to specify the home and storage paths only once throughout the execution of the pipeline, lenstools will do the rest! If you want to build a git repository on top of your simulation batch, you will have to install GitPython and initiate the simulation batch as follows

```
>>> from lenstools.pipeline.remote import LocalGit
>>> batch = SimulationBatch(environment,syshandler=LocalGit())
```

Cosmological parameters

We first need to specify the cosmological model that will regulate the physics of the universe expansion and evolution of the density perturbations. We do this through the cosmology module of astropy (slightly modified to allow the specifications of parameters like n_s and σ_8).

```
>>> from lenstools.pipeline.simulation import LensToolsCosmology
>>> cosmology = LensToolsCosmology(Om0=0.3,Ode0=0.7)
```

The cosmology object will be initialized with $(\Omega_m, \Omega_\Lambda) = (0.3, 0.7)$ and all the other parameters set to their default values

```
>>> cosmology
LensToolsCosmology(H0=72 km / (Mpc s), Om0=0.3, Ode0=0.7, sigma8=0.8, ns=0.96, w0=-1, wa=0, Tcmb0=2.7
```

Now we create a new simulation model that corresponds to the "cosmology" just specified, through our "batch" handler created before

```
>>> model = batch.newModel(cosmology,parameters=["Om","Ol"])

[+] SimTest/Home/Om0.300_Ol0.700 created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700 created on localhost
```

The argument "parameters" specifies which cosmological parameters you want to keep track of in your model; this is useful, for example, when you want to simulate different combinations of these parameters while keeping the other fixed to their default values. The values that are specified in the "parameters" list are translated into the names of the attributes of the `LensToolsCosmology` instance according to the following (customizable) dictionary

```
>>> batch.environment.name2attr

{'Ob': 'Ob0',
'Ol': 'Ode0',
'Om': 'Om0',
'h': 'h',
'ns': 'ns',
'si': 'sigma8',
'w': 'w0',
'wa': 'wa'}
```

Note that lenstools informs you of the directories that are created on disk. You have access at any time to the models that are present in your simulation batch

```
>> batch.models
[<Om=0.300 , Ol=0.700>]
```

Simulation resolution

It is now time to specify the resolution of the N–body simulations that will be run to map the 3D density field of the universe. There are two numbers you need to set here, namely size of the box (that will fix the largest mode your simulations will be able to probe) and the number of particles on a side (that will fix the shortest mode). This command will create a collection of simulations with 512^3 particles in a box of size 240.0 Mpc/h

```
>>> collection = model.newCollection(box_size=240.0*model.Mpc_over_h,nside=512)

[+] SimTest/Home/Om0.300_Ol0.700/512b240 created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/512b240 created on localhost
```

Again, you will have access at any time to the collections that are present in your model

```
>>> model = batch.getModel("Om0.300_Ol0.700")
>>> model.collections

[<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512]
```

Initial conditions

Each simulation collection can have multiple realizations of the density field; these realizations share all the same statistical properties (i.e. the matter power spectrum), but have different spatial arrangements of the particles. This allows you to measure ensemble statistics such as means and covariances of various observables. Let's add three independent realizations of the density field to the "512b240" collection, with random seeds 1,22,333 (the random seed will be used by the initial condition generator to produce different density fields that share the same 3D power spectum)

```
>>> for s in [1,22,333]:
        collection.newRealization(seed=s)

[+] SimTest/Home/Om0.300_Ol0.700/ic1 created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/ic1 created on localhost
[+] SimTest/Home/Om0.300_Ol0.700/ic2 created on localhost
```

```
[+] SimTest/Storage/Om0.300_Ol0.700/ic2 created on localhost
[+] SimTest/Home/Om0.300_Ol0.700/ic3 created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/ic3 created on localhost
```

At this point it should not be surprising that you can do this

```
>>> collection.realizations

[<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=1,seed=1 | IC files on disk: 0 | Snapshot fil
<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=2,seed=22 | IC files on disk: 0 | Snapshot fil
<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=3,seed=333 | IC files on disk: 0 | Snapshot fi
```

Note that, at this step, we are only laying down the directory tree of the simulation batch, and you can see that there are neither IC files nor snapshot files saved on disk yet (this will be produced when we actually run the simulations, but this will be explained later in the tutorial).

Lens planes

For each of the realizations in the collection, we have to create a set of lens planes, that will be necessary for the execution of the ray–tracing step via the multi–lens–plane algorithm. The settings for these lens plane set can be specified through a INI configuration file. Let's call this file "planes.ini"; it should have the following structure

```
[PlaneSettings]

directory_name = Planes
override_with_local = False
format = fits
plane_resolution = 128
first_snapshot = 0
last_snapshot = 58
cut_points = 10.71
thickness = 3.57
length_unit = Mpc
normals = 0,1,2

#Customizable, but optional
name_format = snap{0}_{1}Plane{2}_normal{3}.{4}
snapshots = None
kind = potential
smooth = 1
```

Once you specified the plane configuration file, you can go ahead and create a lens plane set for each of the N–body realizations you created at the previous step

```
>>> from lenstools.pipeline.settings import PlaneSettings
>>> plane_settings = PlaneSettings.read("planes.ini")
>>> for r in collection.realizations:
        r.newPlaneSet(plane_settings)

[+] SimTest/Home/Om0.300_Ol0.700/ic1/Planes created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/ic1/Planes created on localhost
[+] SimTest/Home/Om0.300_Ol0.700/ic2/Planes created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/ic2/Planes created on localhost
[+] SimTest/Home/Om0.300_Ol0.700/ic3/Planes created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/ic3/Planes created on localhost
```

To summarize what you just did, as usual you can type

```
>>> for r in collection.realizations:
        r.planesets

[<Om=0.300 , Ol=0.700>  |  box=240.0 Mpc/h,nside=512  |  ic=1,seed=1  | Plane set: Planes , Plane fil
[<Om=0.300 , Ol=0.700>  |  box=240.0 Mpc/h,nside=512  |  ic=2,seed=22  | Plane set: Planes , Plane fi
[<Om=0.300 , Ol=0.700>  |  box=240.0 Mpc/h,nside=512  |  ic=3,seed=333  | Plane set: Planes , Plane f
```

Weak lensing fields

The last step in the pipeline is to run the multi–lens–plane algorithm through the sets of lens planes just created. This will compute all the ray deflections at each lens crossing and derive the corresponding weak lensing quantities. The ray tracing settings need to be specified in a INI configuration file, that for example we can call "lens.ini". The following configuration will allow you to create square weak lensing simulated maps assuming all the background sources have the same redshift

```
[MapSettings]

directory_name = Maps
override_with_local = False
format = fits
map_resolution = 128
map_angle = 3.5
angle_unit = deg
source_redshift = 2.0

#Random seed used to generate multiple map realizations
seed = 0

#Set of lens planes to be used during ray tracing
plane_set = Planes

#N-body simulation realizations that need to be mixed
mix_nbody_realizations = 1,2,3
mix_cut_points = 0,1,2
mix_normals = 0,1,2
lens_map_realizations = 4

#Which lensing quantities do we need?
convergence = True
shear = True
omega = True

#Customizable, but optional
plane_format = fits
plane_name_format = snap{0}_potentialPlane{1}_normal{2}.{3}
first_realization = 1
```

Different random realizations of the same weak lensing field can be obtained drawing different combinations of the lens planes from different N–body realizations (*mix_nbody_realizations*), different regions of the N–body boxes (*mix_cut_points*) and different rotation of the boxes (*mix_normals*). We create the directories for the weak lensing map set as usual

```
>>> from lenstools.pipeline.settings import MapSettings
>>> map_settings = MapSettings.read("lens.ini")
>>> map_set = collection.newMapSet(map_settings)
```

```
[+] SimTest/Home/Om0.300_Ol0.700/Maps created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/Maps created on localhost
```

And, of course, you can check what you just did

```
>>> collection.mapsets

[<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | Map set: Maps | Map files on disk: 0 ]
```

Now that we layed down our directory tree in a logical and organized fashion, we can proceed with the deployment of the simulation codes. The outputs of these codes will be saved in the "storage" portion of the simulation batch.

5.1.3 Pipeline deployment

After the creation of the directory tree that will host the simulation products (which you can always update calling the appropriate functions on your SimulationBatch instance), it is time to start the production running the actual simulation codes. This implementation of the lensing pipeline relies on three publicly available codes (CAMB , NGenIC and Gadget2) which you have to obtain on your own as the lenstools authors do not own publication rights on them. On the other hand, the lens plane generation and ray–tracing algorithms are part of the lenstools suite. In the remainder of the tutorial, we show how to deploy each step of the pipeline on a computer cluster.

Matter power spectra (CAMB)

The Einstein-Boltzmann code CAMB is used at the first step of the pipeline to compute the matter power spectra that are necessary to produce the initial conditions for the N–body runs. CAMB needs its own parameter file to run, but in order to make things simpler, lenstools provides the *CAMBSettings* class. Typing

```
>>> import lenstools
>>> from lenstools.simulations.camb import CAMBSettings
>>> camb_settings = CAMBSettings()
```

You will have access to the default settings of the CAMB code; you can edit these settings to fit your needs, and then generate the INI parameter file that CAMB will need to run

```
>>> environment = EnvironmentSettings(home="SimTest/Home",storage="SimTest/Storage")
>>> batch = SimulationBatch(environment)
>>> collection = batch.models[0].collections[0]
>>> collection.writeCAMB(z=0.0,settings=camb_settings)

[+] SimTest/Home/Om0.300_Ol0.700/512b240/camb.param written on localhost
```

This will generate a CAMB parameter file that can be used to compute the linear matter power spectrum at redshift $z = 0.0$ (which NGenIC will later scale to the initial redshift of your N–body simulation). You will now need to run the CAMB executable to compute the matter power spectrum as specified by the settings you chose. For how to run CAMB on your computer cluster please refer to the *jobs* section. The basic command you have to run to generate the job submission scripts is, in a shell

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t camb SimTest/Home/collections.txt
```

Initial conditions (NGenIC)

After CAMB finished running, it is time to use the computed matter power spectra to generate the particle displacement field (corresponding to those power spectra) with NGenIC. The NGenIC code needs its own parameter file to run,

which can be quite a hassle to write down yourself. Luckily lenstools provides the *NGenICSettings* class to make things easy:

```
>>> from lenstools.pipeline.settings import NGenICSettings
>>> ngenic_settings = NGenICSettings()
>>> ngenic_settings.GlassFile = lenstools.data("dummy_glass_little_endian.dat")
```

You can modify the attributes of the ngenic_settings object to change the settings to your own needs. There is an additional complication: NGenIC needs the tabulated matter power spectra in a slightly different format than CAMB outputs. Before generating the NGenIC parameter file we will need to make this format connversion

```
>>> collection.camb2ngenic(z=0.0)
[+] CAMB matter power spectrum at SimTest/Home/Om0.300_Ol0.700/512b240/camb_matterpower_z0.000000.txt
```

Next we can generate the NGenIC parameter file

```
>>> for r in collection.realizations:
        r.writeNGenIC(ngenic_settings)

[+] NGenIC parameter file SimTest/Home/Om0.300_Ol0.700/512b240/ic1/ngenic.param written on localhost
[+] NGenIC parameter file SimTest/Home/Om0.300_Ol0.700/512b240/ic2/ngenic.param written on localhost
[+] NGenIC parameter file SimTest/Home/Om0.300_Ol0.700/512b240/ic3/ngenic.param written on localhost
```

For directions on how to run NGenIC on a computer cluster you can refer to the *jobs* section. After the initial conditions files have been produced, you can check that they are indeed present on the storage portion of the directory tree

```
>>> for r in collection.realizations:
        print(r)

<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=1,seed=1 | IC files on disk: 256 | Snapshot fi
<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=2,seed=22 | IC files on disk: 256 | Snapshot f
<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=3,seed=333 | IC files on disk: 256 | Snapshot
```

Note that the IC file count increased from 0 to 256, but the snapshot count is still 0 (because we didn't run Gadget yet). We will explain how to run Gadget2 in the next section. The basic command you have to run to generate the job submission scripts is, in a shell

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t ngenic SimTest/Home/realizations.t
```

Gravitational evolution (Gadget2)

The next step in the pipeline is to run Gadget2 to evolve the initial conditions in time. Again, the Gadget2 tunable settings are handled by lenstools via the `Gadget2Settings`:

```
>>> from lenstools.simulations.gadget2 import Gadget2Settings
>>> gadget_settings = Gadget2Settings()
```

In the gadget_settings instance, you may want to be especially careful in selecting the appropriate values for the OutputScaleFactor and NumFilesPerSnapshot attributes, which will direct which snapshots will be written to disk and in how many files each snapshot will be split. You can generate the Gadget2 parameter file just typing

```
>>> for r in collection.realizations:
        r.writeGadget2(gadget_settings)

[+] Gadget2 parameter file SimTest/Home/Om0.300_Ol0.700/512b240/ic1/gadget2.param written on localhos
[+] Gadget2 parameter file SimTest/Home/Om0.300_Ol0.700/512b240/ic2/gadget2.param written on localhos
[+] Gadget2 parameter file SimTest/Home/Om0.300_Ol0.700/512b240/ic3/gadget2.param written on localhos
```

Now you can submit the Gadget2 runs following the directions in the *jobs* section. The basic command you have to run to generate the job submission scripts is, in a shell

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t gadget2 SimTest/Home/realizations
```

If Gadget2 ran succesfully and produced the required snapshot, this should reflect on your `SimulationIC` instances

```
>>> for r in collection.realizations:
        print(r)

<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=1,seed=1 | IC files on disk: 256 | Snapshot fi
<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=2,seed=22 | IC files on disk: 256 | Snapshot f
<Om=0.300 , Ol=0.700> | box=240.0 Mpc/h,nside=512 | ic=3,seed=333 | IC files on disk: 256 | Snapshot
```

You have access to each of the N–body simulation snapshots through the `Gadget2Snapshot` class.

Lens planes

Now that Gadget2 has finished the execution, we are ready to proceed in the next step in the pipeline. The multi–lens–plane algorithm approximates the matter distribution between the observer and the backround source as a sequence of parallel lens planes with a local surface density proportional to the density constrast measured from the 3D N–body snapshots. lenstools provides an implementation of the density and lensing potential estimation algorithms. You will have to use the same INI configuration file used to create the *planes* section of the directory tree (in the former we called this file "planes.ini"). After filling the appropriate section of "job.ini" as outlined in *jobs* (using "lenstools.planes-mpi" as the executable name), run on the command line

```
lenstools.submission -e SimTest/Home/environment.ini -o planes.ini -j job.ini -t planes SimTest/Home/
```

This will produce the plane generation execution script that, when executed, will submit your job on the queue. If lenstools.planes-mpi runs correctly, you should notice the presence of the new plane files

```
>>> for r in collection.realizations:
        print(r.planesets[0])

<Om=0.300 , Ol=0.700>  |  box=15.0 Mpc/h,nside=32  |  ic=1,seed=1  | Plane set: Planes , Plane files
<Om=0.300 , Ol=0.700>  |  box=15.0 Mpc/h,nside=32  |  ic=2,seed=22  | Plane set: Planes , Plane files
<Om=0.300 , Ol=0.700>  |  box=15.0 Mpc/h,nside=32  |  ic=3,seed=333  | Plane set: Planes , Plane file
```

You can access each plane through the `PotentialPlane` class.

Weak lensing fields γ, κ, ω

Once the lensing potential planes have been created, we are ready for the last step in the pipeline, namely the multi–lens–plane algorithm execution which will produce the simulated weak lensing fields. You will need to use the configuration file "lens.ini" that you used to create the maps section of the directory tree in the weak lensing *fields* section. Here is the relevant extract of the file

```
[MapSettings]

directory_name = Maps
override_with_local = True
format = fits
map_resolution = 128
map_angle = 3.5
angle_unit = deg
source_redshift = 2.0
```

```
#Random seed used to generate multiple map realizations
seed = 0

#Set of lens planes to be used during ray tracing
plane_set = Planes

#N-body simulation realizations that need to be mixed
mix_nbody_realizations = 1,2,3
mix_cut_points = 0,1,2
mix_normals = 0,1,2
lens_map_realizations = 4

#Which lensing quantities do we need?
convergence = True
shear = True
omega = True
```

Note the change "override_with_local=False", which became "override_with_local=True"; this is an optional simplification that you can take advantage of if you want. If this switch is set to true, the ray–tracing script will ignore everyting below the "override_with_local" line and read the remaining options from the "Maps" directory. This is a failsafe that guarantees that the weak lensing fields will be generated using the settings that were originally intended for them, i.e. the ones that you used to create the "Maps" directory in the tree.

After filling the appropriate section of "job.ini" as outlined in *jobs* (using "lenstools.raytracing-mpi" as the executable name), run on the command line

```
lenstools.submission -e SimTest/Home/environment.ini -o lens.ini -j job.ini -t raytracing SimTest/Hor
```

Where "collections.txt", in this case, should be a text file with only one line

```
Om0.300_Ol0.700|512b240
```

After lenstools.raytracing-mpi finished the execution, you will find your weak lensing maps in the "Maps" directory, and you can conveniently access them through the *ConvergenceMap* and *ShearMap* classes.

```
>>> from lenstools import ConvergenceMap
>>> collection.mapsets

[<Om=0.300 , Ol=0.700> | box=15.0 Mpc/h,nside=32 | Map set: Maps | Map files on disk: 12 ]

>>> mp = collection.mapsets[0]
>>> mp.path("WLconv_z2.00_0001r.fits")

"SimTest/Storage/Om0.300_Ol0.700/32b15/Maps/WLconv_z2.00_0001r.fits"

>>> c = ConvergenceMap.load(mp.path("WLconv_z2.00_0001r.fits"))
>>> c.info

Pixels on a side: 128
Pixel size: 98.4375 arcsec
Total angular size: 3.5 deg
lmin=1.0e+02 ; lmax=9.3e+03
```

If you need to generate the weak lensing simulated fields not in image form but in catalog form, you can use the *SimulationCatalog* class instead of the *SimulationMaps* class

```
>>> lenstools.showData("catalog_default.ini")

[CatalogSettings]
```

```
#Name of catalog batch
directory_name = Catalog
input_files = galaxy_positions.fits
total_num_galaxies = 1000
catalog_angle_unit = deg

#Use the options generated at the moment of the batch generation (advised)
override_with_local = True

#Format of the simulated catalog files
format = fits

#Random seed used to generate multiple catalog realizations
seed = 0

#Set of lens planes to be used during ray tracing
plane_set = Planes

#N-body simulation realizations that need to be mixed
mix_nbody_realizations = 1
mix_cut_points = 0
mix_normals = 0
lens_catalog_realizations = 1
```

```
>>> from lenstools.pipeline.settings import CatalogSettings
>>> catalog_settings = CatalogSettings.read(lenstools.data("catalog_default.ini"))
>>> collection.newCatalog(catalog_settings)

[+] SimTest/Home/Om0.300_Ol0.700/Catalog created on localhost
[+] SimTest/Storage/Om0.300_Ol0.700/Catalog created on localhost
```

5.1.4 Computer cluster offload

Generating job submission scripts

Each computer cluster comes with its own computing environment, its own job scheduler and its own job scheduler directives. To accomodate these differences, lenstools provides a platform–independent interface to generate your submission scripts. The job settings are read from a platform–independent INI configuration file, which is passed to a *JobHandler* instance. This job handler instance will translate the user provided settings into the machine specific job directives. This provides a platform–independent job deployment. Here is an example of the job submission options for a Gadget2 run, which we will call "job.ini"

```
[Gadget2]

#Personal settings
email = apetri@phys.columbia.edu
charge_account = TG-AST140041

#Path to executable
path_to_executable = /my/cluster/path/to/the/Gadget2/executable

#Name of the job, output
job_name = Gadget2
redirect_stdout = gadget.out
redirect_stderr = gadget.err
```

```
#Resources
cores_per_simulation = 256
queue = development
wallclock_time = 02:00:00

#Script name
job_script_file = gadget.sh
```

lenstools provides a command line script, lenstools.submission, that will take care of the script generation. The "-s" flag can be used to specify the system we are running on; if not specified, the system is detected automatically looking at the value of the "THIS" environment variable. For example the "-s Stampede" option will generate the submission scripts for the Stampede computer cluster through the *StampedeHandler* job handler. Here it is an example on how the script is generated: from the command line run

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t gadget2 -s Stampede $imTest/Home/r
```

or, if you prefer, lenstools.submission can read from stdin too, and hence you can use shell pipes

```
cat SimTest/Home/realizations.txt | lenstools.submission -e SimTest/Home/environment.ini -j job.ini -
```

In short, the "-e" switch will make sure that we are pointing to the right simulation batch, the "-j" switch will point to the correct platform–independent job option file, the "-t" switch specifies which job submission script we are generating and the realizations.txt file contains a list of the realizations that the script will process. For example if the contents of "realizations.txt" are

```
Om0.300_Ol0.700|512b240|ic1
Om0.300_Ol0.700|512b240|ic2
Om0.300_Ol0.700|512b240|ic3
```

the job submission will process the Om0.300_Ol0.700 model, collection of simulations with 512^3 particles and 240.0Mpc/h box size, initial conditions from 1 to 3. You can additionally specify the –chunks and –one options to change the number of simulations that are processed in parallel.

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t gadget2 -s Stampede $imTest/Home/r
```

will generate 3 job submission scripts, each of which will take care of one of the initial conditions

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t gadget2 -s Stampede $imTest/Home/r
```

will generate one job submission script, in which the 3 initial conditions are processed one after the other, starting with the first. This job will run on 256 cores

```
lenstools.submission -e SimTest/Home/environment.ini -j job.ini -t gadget2 -s Stampede $imTest/Home/r
```

will generate one submission script, in which the 3 initial conditions are processed in parallel. This job will run on 768 cores. This is the output of this execution of lenstools.submission

```
[*] Environment settings for current batch read from SimTest/Home/environment.ini
[+] Using job handler for system Stampede
[*] Current batch home directory: SimTest/Home
[*] Current batch mass storage: SimTest/Storage
[*] Realizations to include in this submission will be read from realizations.txt
[+] Found 3 realizations to include in job submission, to be split in 1 chunks
[+] Generating Gadget2 submission script
[*] Reading job specifications from jobs.ini section Gadget2
[+] Stdout will be directed to SimTest/Home/Logs/gadget.out
[+] Stderr will be directed to SimTest/Home/Logs/gadget.err
[+] SimTest/Home/Jobs/gadget1.sh written on localhost
```

On Stampede you submit the jobs to the queue using the "sbatch" command:

```
sbatch SimTest/Home/Jobs/gadget1.sh
```

Generic job submissions

lenstools provides functionality to distribute execution of arbitrary code throughout all your simulation batch. Suppose that you compiled an executable "myexec" for your own purposes; if this executable accepts the "-e" and "-c" options, i.e. you can run it like this

```
mpiexec -n 16 ./myexec -e SimTest/Home/environment.ini -c code_options.ini "Om0.300_0.700|512b240|ic1
```

Then lenstools.submission can help you distribute the myexec execution across your simulation batch: you just have to include the following section in your "job.ini"

```
[/path/to/myexec]

#Personal settings
email = apetri@phys.columbia.edu
charge_account = TG-AST140041

#Name of the job, output
job_name = myexecJob
redirect_stdout = myexec.out
redirect_stderr = myexec.err

#Resources
cores_per_simulation = 16
queue = development
wallclock_time = 02:00:00

#Script name
job_script_file = myexec.sh
```

And, in a shell, type

```
lenstools.submission -e SimTest/Home/environment.ini -o code_options.ini -j job.ini -t "/path/to/myex
```

to generate the submission script.

Job handlers for different clusters

Each computer cluster comes with its own job sheduler and job submission directives. lenstools facilitates the transition between clusters by translating the platform–independent options contained in "job.ini" into cluster specific directives through the *JobHandler* objects. Currently the "-s" switch that you can pass to lenstools.submission accepts the values "Stampede" (that will select the *StampedeHandler* handler) and "edison" (that will select *EdisonHandler*). Should you want to use a different computer cluster, this is what you have to do. Create a file called mycluster.py, and implement a class MyCluster as follows (this is just an example)

```
#mycluster.py

from lenstools.pipeline.deploy import JobHandler,Directives,ClusterSpecs
import astropy.units as u

_SLURMspecs = {
"directive_prefix" : "#SBATCH",
"charge_account_switch" : "-A ",
```

```
"job_name_switch" : "-J ",
"stdout_switch" : "-o ",
"stderr_switch" : "-e ",
"num_cores_switch" : "-n ",
"num_nodes_switch" : "-N ",
"tasks_per_node_switch" : None,
"queue_type_switch" : "-p ",
"wallclock_time_switch" : "-t ",
"user_email_switch" : "--mail-user=",
"user_email_type" : "--mail-type=all",
}

_MyClusterSpecs = {
"shell_prefix" : "#!/bin/bash",
"execution_preamble" : None,
"job_starter" : "ibrun",
"cores_per_node" : 16,
"memory_per_node" : 32.0*u.Gbyte,
"cores_at_execution_switch" : "-n ",
"offset_switch" : "-o ",
"wait_switch" : "wait",
"multiple_executables_on_node" : True
}

class MyCluster(JobHandler):

        """
        Job handler for my cluster

        """

        def setDirectives(self):
                self._directives = Directives(**_SLURMspecs)

        def setClusterSpecs(self):
                self._cluster_specs = ClusterSpecs(**_MyClusterSpecs)
```

After doing this, you just need to pass the string "mycluster.MyCluster" to the "-s" switch when you run lenstools.submission and you are all set!

5.1.5 Post processing

This section shows an example on how to do some post processing on the products of your simulation batch (for example measuring the N–body simulations power spectra). The basic idea is to define a function with the signature

```
>>> def methodThatMeasuresSomething(pool,batch,settings,id,**kwargs):
        ...
```

where

- pool is a `MPIWhirlPool` instance that will take care of the parallelization of the code
- batch is the simulation batch object, i.e. an instance of `SimulationBatch`
- settings are the tunable settings of the code
- id is the particular batch subset to process, for example "Om0.300_Ol0.700|512b240|ic1"
- kwargs are any other keyword arguments you may want to pass to the methodThatMeasuresSomething method

lenstools will take care of distributing the methodThatMeasuresSomething calls on the computer cluster you are running on. Below is a working example of how to measure the 3D matter power spectrum out of the simulation boxes.

Example: measure the 3D matter power spectrum

Create a file "matter_power_spectrum.py"

```
##############################################################################
###############Measure statistics out of N-body simulation snapshots#############
##############################################################################

import sys,os
import logging

from distutils import config
from ConfigParser import NoOptionError

from lenstools.utils import MPIWhirlPool

from lenstools.simulations.nbody import NbodySnapshot
from lenstools.simulations.gadget2 import Gadget2Snapshot

from lenstools.pipeline.simulation import SimulationBatch

import numpy as np
import astropy.units as u

################################################
###########Loggers##############################
################################################

console = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s:%(name)-12s:%(levelname)-4s: %(message)s",datefmt='%m-%d %
console.setFormatter(formatter)

logdriver = logging.getLogger("lenstools.driver")
logdriver.addHandler(console)
logdriver.propagate = False

#Orchestra director of the execution
def powerSpectrumExecution():

        script_to_execute = matterPowerSpectrum
        settings_handler = PowerSpectrumSettings
        kwargs = {"fmt":Gadget2Snapshot}

        return script_to_execute,settings_handler,kwargs

##############################################################
###############Snapshot power spectrum#########################
##############################################################

def matterPowerSpectrum(pool,batch,settings,id,**kwargs):

        assert "fmt" in kwargs.keys()
        fmt = kwargs["fmt"]

        #Safety type check
```

```
        assert isinstance(pool,MPIWhirlPool) or (pool is None)
        assert isinstance(batch,SimulationBatch)
        assert isinstance(fmt(),NbodySnapshot)
        assert isinstance(settings,PowerSpectrumSettings)

        #Split the id into the model,collection and realization parts
        cosmo_id,geometry_id = id.split("|")

        #Get a handle on the simulation model
        model = batch.getModel(cosmo_id)

        #Scale the box size to the correct units
        nside,box_size = geometry_id.split("b")
        box_size = float(box_size)*model.Mpc_over_h

        #Get the handle on the collection
        collection = model.getCollection(box_size,nside)

        #Log the power spectrum settings to the user
        if (pool is None) or (pool.is_master()):

                logdriver.info("Measuring power spectrum for Ensemble {0}".format(settings.ensemble_n
                logdriver.info("The Ensemble will be built with the following N-body realizations: {0
                logdriver.info("First snapshot: {0}".format(settings.first_snapshot))
                logdriver.info("Last snapshot: {0}".format(settings.last_snapshot))
                logdriver.info("Minimum wavenumber: {0}".format(settings.kmin.to(model.Mpc_over_h**-1
                logdriver.info("Maximum wavenumber: {0}".format(settings.kmax.to(model.Mpc_over_h**-1
                logdriver.info("Bin size: {0}".format(((settings.kmax-settings.kmin)/settings.num_k_b
                logdriver.info("FFT grid size: {0}".format(settings.fft_grid_size))
                logdriver.info("Number of bins: {0}".format(settings.num_k_bins))

                #Create dedicated ensemble directory
                ensemble_dir = os.path.join(collection.home_subdir,settings.ensemble_name)
                if not os.path.isdir(ensemble_dir):
                        os.mkdir(ensemble_dir)

        #Construct the array of bin edges
        k_egdes  = np.linspace(settings.kmin,settings.kmax,settings.num_k_bins+1).to(model.Mpc_over_h

        #Cycle over snapshots
        for n in range(settings.first_snapshot,settings.last_snapshot+1):

                #Allocate memory for the power spectrum ensemble
                power_ensemble = np.zeros((len(settings.nbody_realizations),settings.num_k_bins)) *

                #Log to user
                if (pool is None) or (pool.is_master()):
                        logdriver.info("Processing snapshot {0} of model {1}".format(n,id))
                        logdriver.info("Allocated memory for power spectrum Ensemble {0}".format(powe

                for r,ic in enumerate(settings.nbody_realizations):

                        #Log to user
                        if (pool is None) or (pool.is_master()):
                                logdriver.info("Processing N-body realization {0}".format(ic))

                        #Instantiate the appropriate SimulationIC object
                        realization = collection.getRealization(ic)
```

```
                        #Open the snapshot, measure the power spectrum and close it
                        if pool is not None:
                                if realization.gadget_settings.NumFilesPerSnapshot!=pool.size+1:
                                        logdriver.error("The number of snapshots written in parallel
                                        sys.exit(1)

                        snap = fmt.open(realization.snapshotPath(n,sub=None),pool=pool)
                        k,power_ensemble[r],hits = snap.powerSpectrum(k_egdes,resolution=settings.fft
                        snap.close()

                        #Safety barrier sync
                        if pool is not None:
                                pool.comm.Barrier()

                #Save the bin edges and mode counts
                if n==settings.first_snapshot and (pool is None or pool.is_master()):

                        savename = os.path.join(collection.home_subdir,settings.ensemble_name,setting
                        logdriver.info("Saving wavevectors ({0}) to {1}".format(k.unit.to_string(),sa
                        np.save(savename,k.value)

                        savename = os.path.join(collection.home_subdir,settings.ensemble_name,setting
                        logdriver.info("Saving number of modes to {0}".format(savename))
                        np.save(savename,hits)

                #Save the ensemble
                if (pool is None) or (pool.is_master()):

                        savename = os.path.join(collection.home_subdir,settings.ensemble_name,setting
                        logdriver.info("Saving power spectrum Ensemble ({0}) to {1}".format(power_ens
                        np.save(savename,power_ensemble.value)


                #Safety barrier sync
                if pool is not None:
                        pool.comm.Barrier()


        #Completed
        if pool is None or pool.is_master():
                logdriver.info("DONE!!")



##########################################################
###########PowerSpectrumSettings class###################
##########################################################

class PowerSpectrumSettings(object):

        """
        Class handler of N-Body simulation power spectrum measurement settings

        """

        def __init__(self,**kwargs):

                #Tunable settings (resolution, etc...)
```

```
                self.ensemble_name = "gadget2_ps"
                self.nbody_realizations = [1]
                self.first_snapshot = 46
                self.last_snapshot = 58
                self.fft_grid_size = 256
                self.kmin = 0.003 * u.Mpc**-1
                self.kmax = 1.536 * u.Mpc**-1
                self.length_unit = u.Mpc
                self.num_k_bins = 50

                #Allow for kwargs override
                for key in kwargs.keys():
                        setattr(self,key,kwargs[key])

        @classmethod
        def read(cls,config_file):

                #Read the options from the ini file
                options = config.ConfigParser()
                options.read([config_file])

                #Check that the config file has the appropriate section
                section = "PowerSpectrumSettings"
                assert options.has_section(section),"No {0} section in configuration file {1}".format

                #Fill in the appropriate fields
                settings = cls()

                settings.ensemble_name = options.get(section,"ensemble_name")

                #Read in the nbody realizations that make up the ensemble
                settings.nbody_realizations = list()
                for r in options.get(section,"nbody_realizations").split(","):

                        try:
                                l,h = r.split("-")
                                settings.nbody_realizations.extend(range(int(l),int(h)+1))
                        except ValueError:
                                settings.nbody_realizations.extend([int(r)])

                settings.first_snapshot = options.getint(section,"first_snapshot")
                settings.last_snapshot = options.getint(section,"last_snapshot")

                settings.fft_grid_size = options.getint(section,"fft_grid_size")

                settings.length_unit = getattr(u,options.get(section,"length_unit"))
                settings.kmin = options.getfloat(section,"kmin") * settings.length_unit**-1
                settings.kmax = options.getfloat(section,"kmax") * settings.length_unit**-1

                settings.num_k_bins = options.getint(section,"num_k_bins")

                #Return to user
                return settings
```

Create a INI configuration file “code_options.ini”:

```
[PowerSpectrumSettings]
```

```
ensemble_name = gadget2_ps
nbody_realizations = 1,2-3
first_snapshot = 46
last_snapshot = 47
fft_grid_size = 64
kmin = 0.06
kmax = 5.0
length_unit = Mpc
num_k_bins = 10
```

You deploy like this

```
lenstools.execute-mpi -e SimTest/Home/environment.ini -c code_options.ini -m matter_power_spectrum.po
```

And this is an example output

```
04-21 17:32:lenstools.preamble:INFO: Importing lenstools.scripts.nbody.powerSpectrumExecution
04-21 17:32:lenstools.preamble:INFO: Executing: matterPowerSpectrum()
04-21 17:32:lenstools.preamble:INFO: Job configuration handler: PowerSpectrumSettings
04-21 17:32:lenstools.preamble:INFO: Keyword arguments: {'fmt': <class 'lenstools.simulations.gadget2
04-21 17:32:lenstools.preamble:INFO: Reading environment from SimTest/environment.ini
04-21 17:32:lenstools.preamble:INFO: Reading job configuration from code_options.ini
04-21 17:32:lenstools.driver:INFO: Measuring power spectrum for Ensemble gadget2_ps
04-21 17:32:lenstools.driver:INFO: The Ensemble will be built with the following N-body realizations:
04-21 17:32:lenstools.driver:INFO: First snapshot: 46
04-21 17:32:lenstools.driver:INFO: Last snapshot: 47
04-21 17:32:lenstools.driver:INFO: Minimum wavenumber: 0.0833333333333 1 / Mpc/h
04-21 17:32:lenstools.driver:INFO: Maximum wavenumber: 6.94444444444 1 / Mpc/h
04-21 17:32:lenstools.driver:INFO: Bin size: 0.686111111111 1 / Mpc/h
04-21 17:32:lenstools.driver:INFO: FFT grid size: 64
04-21 17:32:lenstools.driver:INFO: Number of bins: 10
04-21 17:32:lenstools.driver:INFO: Processing snapshot 46 of model Om0.300_Ol0.700|512b240
04-21 17:32:lenstools.driver:INFO: Allocated memory for power spectrum Ensemble (2, 10)
04-21 17:32:lenstools.driver:INFO: Processing N-body realization 1
04-21 17:32:lenstools.driver:INFO: Processing N-body realization 2
04-21 17:32:lenstools.driver:INFO: Processing N-body realization 3
04-21 17:32:lenstools.driver:INFO: Saving wavevectors (1 / Mpc/h) to SimTest/Home/Om0.300_Ol0.700/512
04-21 17:32:lenstools.driver:INFO: Saving number of modes to SimTest/Home/Om0.300_Ol0.700/512b240/gad
04-21 17:32:lenstools.driver:INFO: Saving power spectrum Ensemble (Mpc/h3) to SimTest/Home/Om0.300_Ol
04-21 17:32:lenstools.driver:INFO: Processing snapshot 47 of model Om0.300_Ol0.700/512b240|512b240
04-21 17:32:lenstools.driver:INFO: Allocated memory for power spectrum Ensemble (2, 10)
04-21 17:32:lenstools.driver:INFO: Processing N-body realization 1
04-21 17:32:lenstools.driver:INFO: Processing N-body realization 2
04-21 17:32:lenstools.driver:INFO: Processing N-body realization 3
04-21 17:32:lenstools.driver:INFO: Saving power spectrum Ensemble (Mpc/h3) to SimTest/Home/Om0.300_Ol
04-21 17:32:lenstools.driver:INFO: DONE!!
```

5.1.6 Default settings

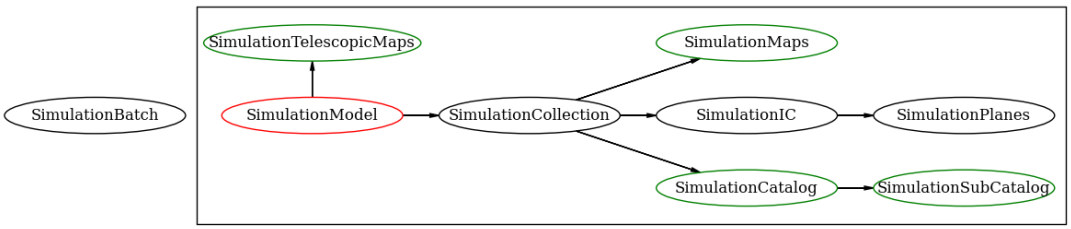
You can visualize the default INI configuration files for the different steps in the pipeline by typing in a python shell

```
import lenstools

#Default job submission
lenstools.showData("job_default.ini")

#Default lensing options
```

```
lenstools.showData("lens_default.ini")

#Default telescopic lensing options
lenstools.showData("telescopic_default.ini")

#Default catalog production options
lenstools.showData("catalog_default.ini")
```

5.1.7 Class inheritance

This is a simplifying scheme of the class inheritance used in the lenstools pipeline



5.2 Ray tracing simulations

lenstools provides an implementation of the multi–lens–plane algorithm. For a hands-on demo of the current capabilities of the ray tracer, please look at this notebook

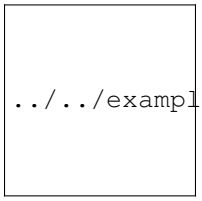
5.3 Existing Weak Lensing simulation suites

At the current stage of Weak gravitational lensing research, large numerical simulations are required for analyzing observations; the LensTools python package provides an API to interact with some already existing simulated datasets (mainly convergence and shear maps for different cosmological models), such as

1. The IGS1 simulations: this simulated dataset contains 1000 realizations of single redshift convergence and shear maps for six different cosmological parammeter combinations (a fiducial model and some variations of the quadruplet $(\Omega_m, w, \sigma_8, n_s)$). The fiducial model is based on 45 independent N-body simulations and the variations are based on 5 independent N-body simulations (where $N = 512^3$)
2. The CFHTemu1 simulations: this simulated dataset contains 1000 realizations of convergence maps with the source redshift distribution of the CFHTLens survey; the simulated foregrounds are available for 91 different cosmological parameter variations of the triplet (Ω_m, w, σ_8)

This is an example on how you can use the LensTools API to interact with the simulations: suppose you have a local copy of the IGS1 simulations, which you saved in '/user/igs1' and you didn't modify the original directory tree. Then here is how you can interact with the maps

```
>>> from lenstools.simulations import IGS1

#Instantiate one of the models, for example the reference one
>>> igs1_ref = IGS1(root_path="/user/igs1",H0=72.0,Om0=0.26,sigma8=0.798,ns=0.96,w0=-1)

#Now you can access the names of the individual realizations
>>> igs1_ref.getNames(2,z=1.0)

/user/igs1/m-512b240_Om0.260_Ol0.740_w-1.000_ns0.960_si0.798/Maps/WL-conv_m-512b240_Om0.260_Ol0.740_w

>>> igs1_ref.getNames(3,z=1.0,big_fiducial_set=True)

/user/igs1/m-512b240_Om0.260_Ol0.740_w-1.000_ns0.960_si0.798_f/Maps/WL-conv_m-512b240_Om0.260_Ol0.740

#Or you can load the images directly in memory
>>> image = igs1_ref.load(2,z=1.0)

#This wraps the image in a ConvergenceMap instance, now you can, for example, display some info and v
>>> image.info

Pixels on a side: 2048
Pixel size: 6.08923828125 arcsec
Total angular size: 3.4641 deg
lmin=1.0e+02 ; lmax=1.5e+05

>>> image.visualize()
```



Read more in the API section to learn what you can do with ConvergenceMap instances; while LensTools provides the API to interact with a local copy of these simulated datasets, it does not provide the datasets themselves; to obtain the IGS1 dataset please email Jan M. Kratochvil, to obtain the CFHTemu1 dataset please email Andrea Petri

CHAPTER 6

Command line scripts

6.1 LensTools command line scripts

6.1.1 General purpose scripts

gadgetheader

Displays the header information of a Gadget2 snapshot. Usage:

```
gadgetheader <file_1> <file_2> ...
```

gadgetstrip

Strips the sometimes unnecessary velocity information from a Gadget2 snapshot, roughly reducing its disk usage by a factor of 2. Usage:

```
gadgetstrip <file_1> <file_2> ...
```

The script can also read from standard input: to strip all the files in a directory:

```
ls | gadgetstrip
```

npyinfo

lenstools.confidencecontour

lenstools.view

6.1.2 LensTools pipeline scripts

lenstools.submission

lenstools.cutplanes

lenstools.cutplanes-mpi

lenstools.raytracing

lenstools.raytracing-mpi

lenstools.execute-mpi

CHAPTER 7

Gallery

7.1 Gallery

7.1.1 Measure the power spectrum of a convergence map

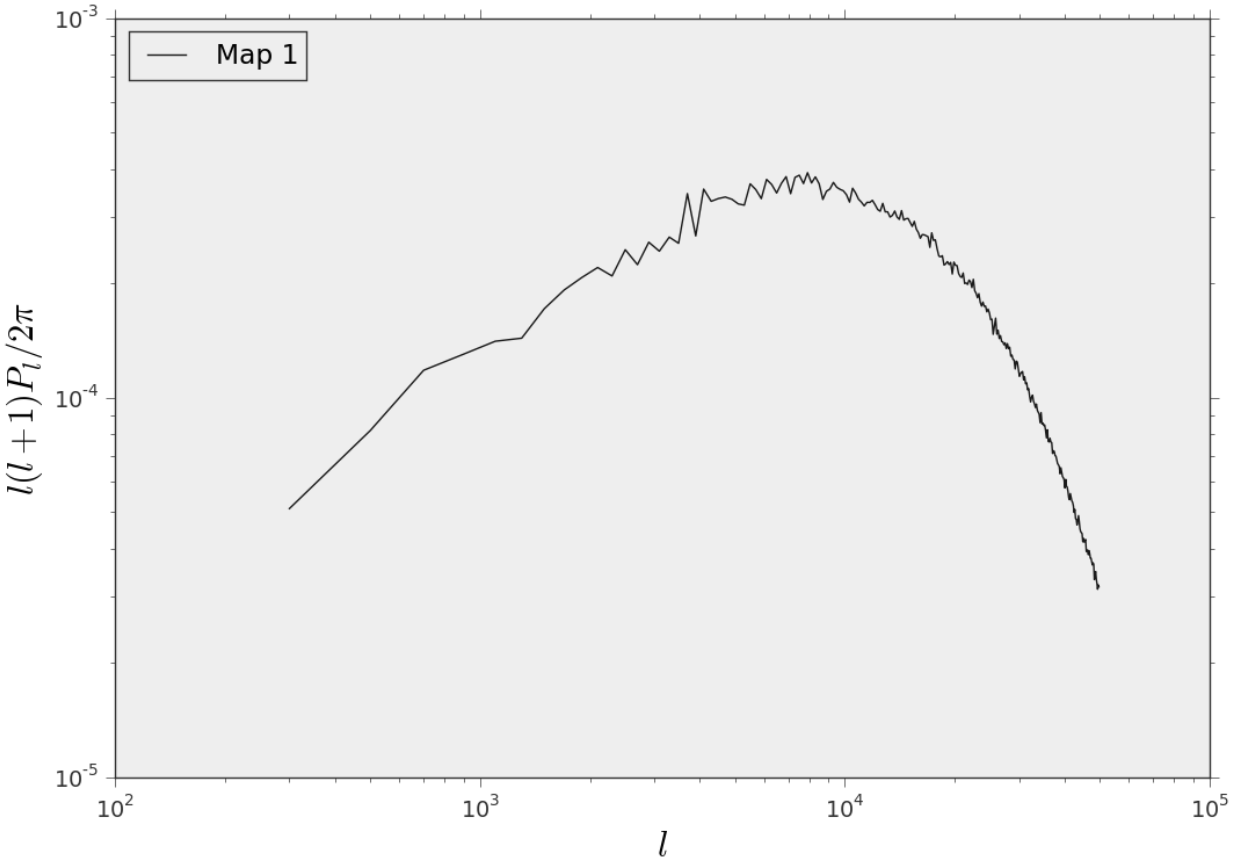
The code you have to run to measure the power spectrum of a convergence map looks something like this

```
#The operations on convergence maps are handled with the ConvergenceMap class
from lenstools import ConvergenceMap

import numpy as np
import matplotlib.pyplot as plt

map_filename = "Data/conv1.fit"
conv_map = ConvergenceMap.load(map_filename)

l_edges = np.arange(200.0,50000.0,200.0)

#Measure the power spectrum calling the powerSpectrum method
l,Pl = conv_map.powerSpectrum(l_edges)

#Plot the result
fig,ax = plt.subplots()
ax.plot(l,l*(l+1)*Pl/(2.0*np.pi),label="Map 1")
ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel(r"$l$")
ax.set_ylabel(r"$l(l+1)P_l/2\pi$")
ax.legend(loc="upper left")

fig.savefig("power_spectrum_one_map.png")
```

And this is the result

7.1.2 Measure the power spectrum of a convergence map (MPI)

When you need to measure the power spectrum of many different convergence maps (for example 1000 different realizations of a field of view), you are effectively measuring an Ensemble of quantities. lenstools provides Ensemble computing tools through the *Ensemble* class. This class supports parallel operations too through MPI, like in this example. First define a function that measures the power spectrum out of a single map:


Map 1
10^{-3}
10^{-4}
10^{-5}
$l(l+1)P_l/2\pi$
10^2
10^3
10^4
10^5
l

```
#The operations on convergence maps are handled with the ConvergenceMap class
from lenstools import ConvergenceMap

def measure_power_spectrum(filename,l_edges):

        conv_map = ConvergenceMap.load(filename)
        l,Pl = conv_map.powerSpectrum(l_edges)
        return Pl
```

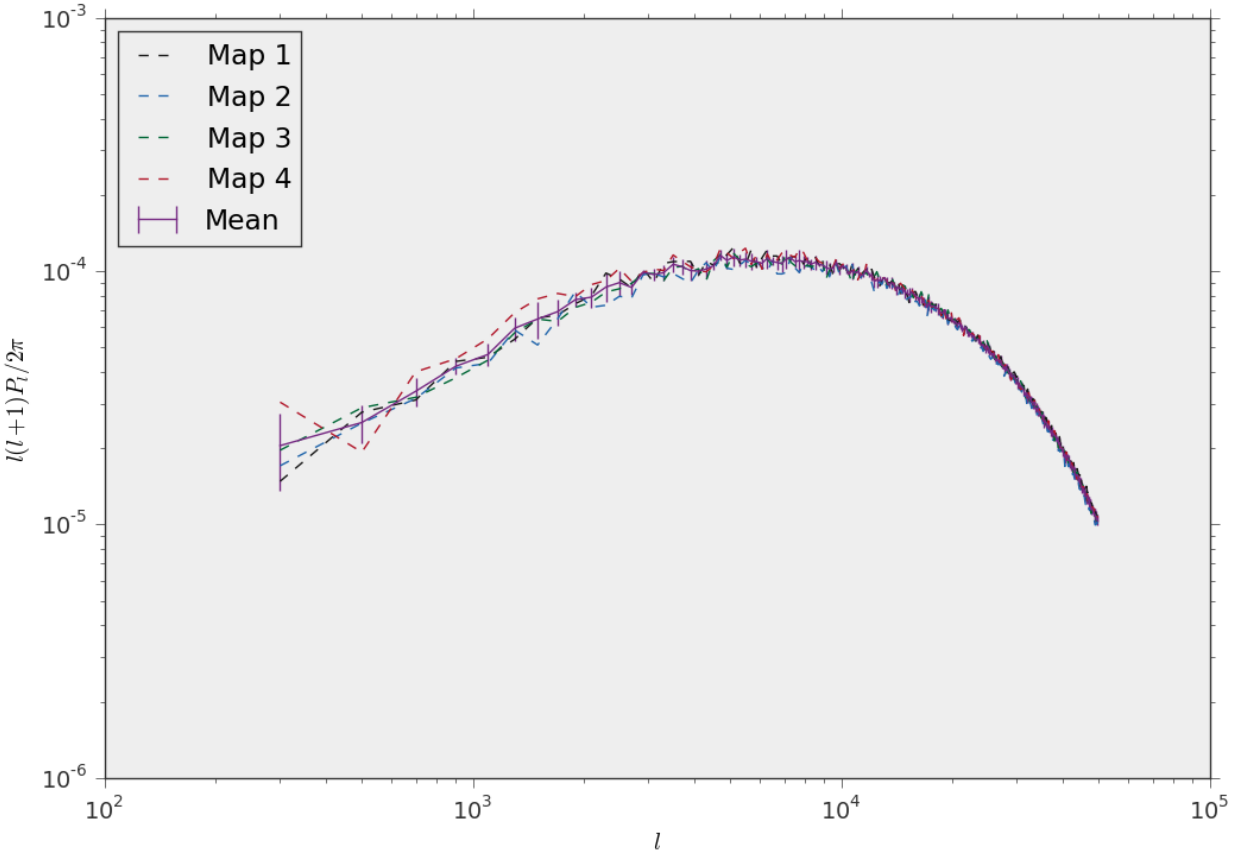
This is the actual code:

```
from lenstools.statistics.ensemble import Ensemble
from lenstools.utils.decorators import Parallelize

import logging

import numpy as np
import matplotlib.pyplot as plt

logging.basicConfig(level=logging.DEBUG)

@Parallelize.masterworker
def main(pool):

        l_edges = np.arange(200.0,50000.0,200.0)
        l = 0.5*(l_edges[:-1] + l_edges[1:])

        conv_ensemble = Ensemble.compute(["Data/conv1.fit","Data/conv2.fit","Data/conv3.fit","Data/co

        fig,ax = plt.subplots()
        for n in range(len(conv_ensemble)):
                ax.plot(l,l*(l+1)*conv_ensemble.iloc[n]/(2.0*np.pi),label="Map {0}".format(n+1),lines

        mean = conv_ensemble.mean(0)
        errors = np.sqrt(conv_ensemble.covariance().values.diagonal())

        ax.errorbar(l,l*(l+1)*mean/(2.0*np.pi),yerr=l*(l+1)*errors/(2.0*np.pi),label="Mean")

        ax.set_xscale("log")
        ax.set_yscale("log")
        ax.set_xlabel(r"$l$")
        ax.set_ylabel(r"$l(l+1)P_l/2\pi$")
        ax.legend(loc="upper left")

        plt.savefig("power_ensemble.png")

if __name__=="__main__":
        main(None)
```

And this is the result

7.1.3 Distribute operations across different processors

You can take advantage of the *Ensemble* class to distribute custom operations (that do not require inter–process communication) across different processors. Suppose we want to compute the square of a bunch of numbers contained in a file and save the result to another file. Suppose also we want to do this on 100 different files and spread the computations across independent processes. The following code will do it:


Map 1
Map 2
Map 3
Map 4
Mean
10^{-3}
10^{-4}
10^{-5}
10^{-6}
$l(l+1)P_l/2\pi$
10^2
10^3
10^4
10^5
l

```
import numpy as np
from lenstools.utils.decorators import Parallelize
from lenstools.statistics.ensemble import Ensemble

#Define the function that computes the square
def square(filename):

        data = np.loadtxt(filename)
        try:
                np.savetxt("square_"+filename,data**2)
                return 0
        except IOError:
                return 1

#Main execution
@Parallelize.masterworker
def main(pool):

        #Operate on these files
        filelist = [ "file{0}.txt".format(n+1) for n in range(100) ]

        #ens will be an instance of :py:class:`~lenstools.statistics.ensemble.Ensemble` with only one
        ens = Ensemble.compute(filelist,callback_loader=square,pool=pool)

if __name__=="__main__":
        main(None)
```

7.1.4 Histograms of convergence maps

```
import sys

#####################################################
#######LensTools functionality#######################
#####################################################

from lenstools import ConvergenceMap,Ensemble,GaussianNoiseGenerator
from lenstools.statistics.index import PDF,Indexer
from lenstools.utils.defaults import load_fits_default_convergence
from lenstools.simulations import IGS1

#####################################################

import numpy as np
from astropy.units import deg,arcmin
import matplotlib.pyplot as plt

from emcee.utils import MPIPool

import logging
import argparse

#############################################################################################
#########This function gets called on every map image and computes the histograms########
#############################################################################################

def compute_histograms(map_id,simulation_set,smoothing_scales,index,generator,bin_edges):
```

```
        assert len(index.descriptor_list) == len(smoothing_scales)

        z = 1.0

        #Get map name to analyze
        map_name = simulation_set.getNames(z=z,realizations=[map_id])[0]

        #Load the convergence map
        convergence_map = ConvergenceMap.load(map_name)

        #Generate the shape noise map
        noise_map = generator.getShapeNoise(z=z,ngal=15.0*arcmin**-2,seed=map_id)

        #Add the noise
        convergence_map += noise_map

        #Measure the features
        hist_output = np.zeros(index.size)
        for n,descriptor in enumerate(index.descriptor_list):

                logging.debug("Processing {0} x {1}".format(map_name,smoothing_scales[n]))

                smoothed_map = convergence_map.smooth(smoothing_scales[n])
                v,hist_output[descriptor.first:descriptor.last] = smoothed_map.pdf(bin_edges)

        #Return the histograms in array format
        return hist_output

##############################################################
########################Main loop#############################
##############################################################

if __name__=="__main__":

        #Initialize MPI pool
        try:
                pool = MPIPool()
        except ValueError:
                pool = None

        #Parse command line arguments
        parser = argparse.ArgumentParser()
        parser.add_argument("-v","--verbose",action="store_true",default=False,dest="verbose",help="I
        parser.add_argument("-p","--path",action="store",dest="path",default="/Users/andreapetri/Docu
        parser.add_argument("-n","--num_realizations",dest="num_realizations",action="store",type=int

        cmd_args = parser.parse_args()

        #Set logging level
        if cmd_args.verbose:
                logging.basicConfig(level=logging.DEBUG)
        else:
                logging.basicConfig(level=logging.INFO)

        if (pool is not None) and not(pool.is_master()):

                pool.wait()
                sys.exit(0)
```

```
        #Root path of IGS1 maps
        root_path = cmd_args.path
        num_realizations = cmd_args.num_realizations

        #Smoothing scales in arcmin
        smoothing_scales = [ theta*arcmin for theta in [0.1,0.5,1.0,2.0] ]
        bin_edges = np.ogrid[-0.15:0.15:128j]
        bin_midpoints = 0.5*(bin_edges[1:] + bin_edges[:-1])

        #Create smoothing scale index for the histogram
        idx = Indexer.stack([PDF(bin_edges) for scale in smoothing_scales])

        #Create IGS1 simulation set object to look for the right simulations
        simulation_set = IGS1(root_path=root_path)

        #Look at a sample map
        sample_map = ConvergenceMap.load(simulation_set.getNames(z=1.0,realizations=[1])[0])

        #Initialize Gaussian shape noise generator
        generator = GaussianNoiseGenerator.forMap(sample_map)

        #Build Ensemble instance with the maps to analyze
        map_ensemble = Ensemble.fromfilelist(range(1,num_realizations+1))

        #Measure the histograms and load the data in the ensemble
        map_ensemble.load(callback_loader=compute_histograms,pool=pool,simulation_set=simulation_set,

        if pool is not None:
                pool.close()

        ###############################################################################################
        ##############################Ensemble data available at this point for covariance, PCA, etc
        ###############################################################################################

        #Plot results to check
        fig,ax = plt.subplots(len(smoothing_scales),1)
        for i in range(len(smoothing_scales)):

                mean = map_ensemble.mean()[idx[i].first:idx[i].last]
                error = np.sqrt(map_ensemble.covariance().diagonal()[idx[i].first:idx[i].last])

                ax[i].errorbar(bin_midpoints,mean,yerr=error)

                ax[i].set_xlabel(r"$\kappa$")
                ax[i].set_ylabel(r"$P(\kappa)$")
                ax[i].set_title(r"${0:.1f}^\prime={1:.1f}$pix".format(smoothing_scales[i].value,(smoo


        fig.tight_layout()
        fig.savefig("histograms.png")
```

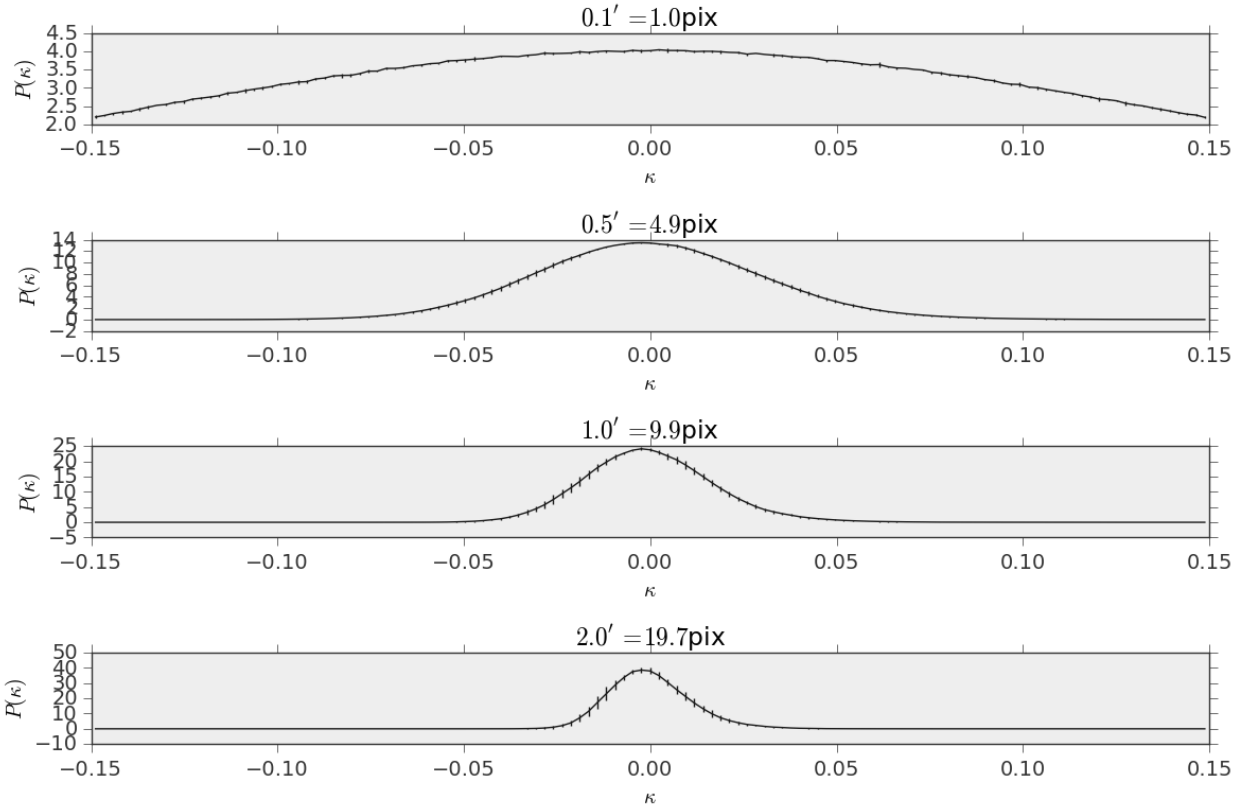
You run this typing:

```
python histograms.py -p <path_to_your_simulations> -n <number_of_realizations>
```

Or, if you have a MPI installation and want to run on multiple processors:

```
mpiexec -n <number_of_processors> python histograms.py -p <path_to_your_simulations> -n <number_of_re
```

This is how the result looks like

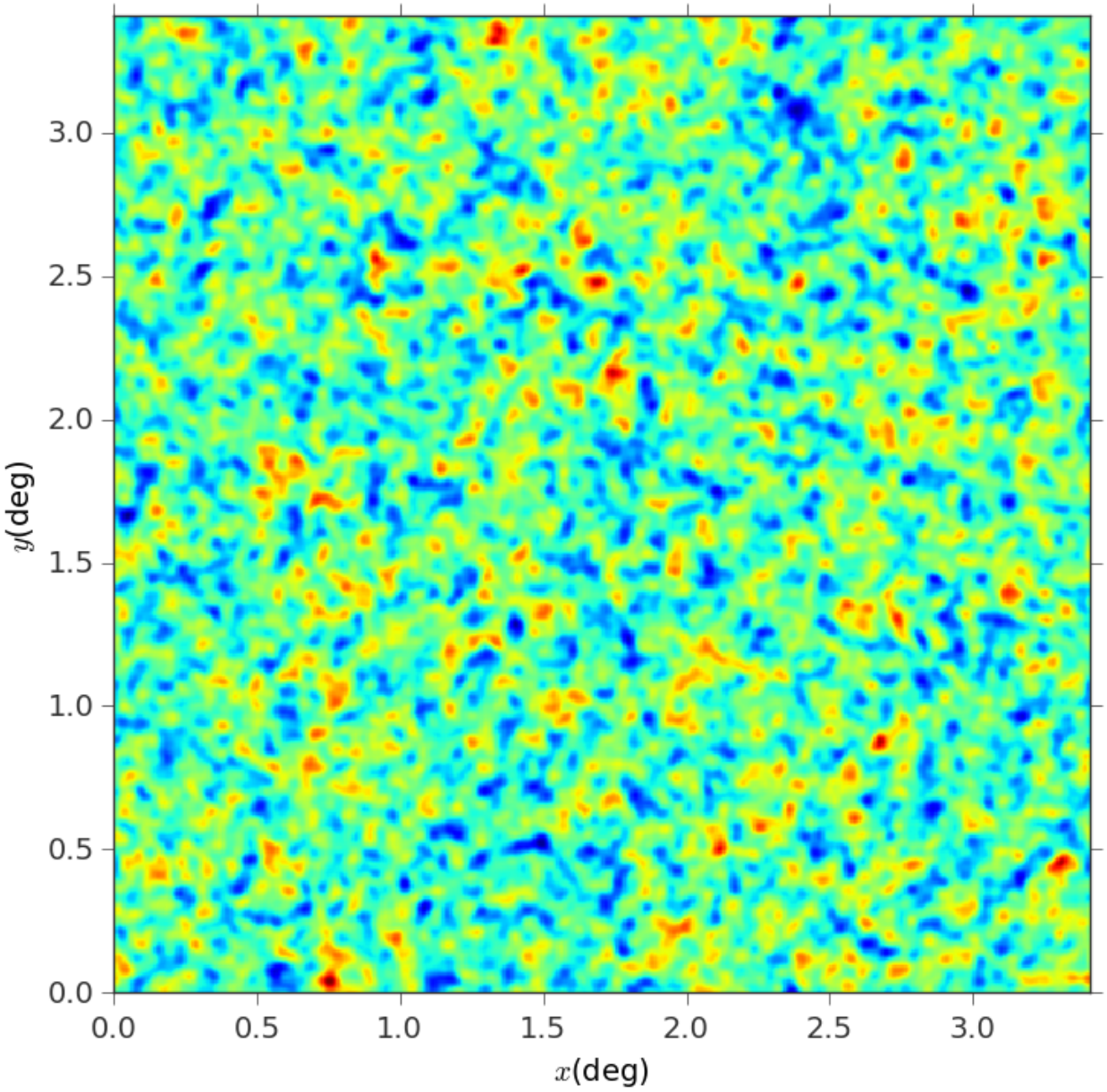


7.1.5 Generate a realization of a gaussian random field with known power spectrum

```
"""
Generate gaussian random fields with a known power spectrum

"""

import numpy as np
import matplotlib.pyplot as plt

from astropy.units import deg

from lenstools import GaussianNoiseGenerator

#Set map side angle, and number of pixels on a side
num_pixel_side = 512
side_angle = 3.41 * deg

#Read the power spectrum (l,Pl) from an external file, and load it in numpy array format (the generat
l,Pl = np.loadtxt("Data/ee4e-7.txt",unpack=True)

#Instantiate the gaussian noise generator
gen = GaussianNoiseGenerator(shape=(num_pixel_side,num_pixel_side),side_angle=side_angle,label="conve
```

```
#Generate one random realization
gaussian_map = gen.fromConvPower(np.array([l,Pl]),seed=1,kind="linear",bounds_error=False,fill_value=

#gaussian_map is a ConvergenceMap instance
gaussian_map.visualize()
gaussian_map.savefig("example_map.png")
```

This is the result



7.1.6 Construct convergence maps out of shear catalogs

Constructing a convergence map of a particular subfield from a shear catalog is an operation that occurs frequently in weak lensing analysis; if the field of view is unmasked, the reconstruction operation is equivalent to the calculation of

the E mode of the shear field γ. Here is how to to use lenstools to perform this operation: the operations are handled with the *ShearCatalog* class

```
from lenstools.catalog.shear import ShearCatalog
import astropy.table as tbl
import astropy.units as u
```

Suppose that the shear table is contained in a file called 'WLshear.fits' (which should contain a table with columns 'shear1','shear2' readable with astropy.table.Table.read()), in which each row represents a different galaxy; suppose also that the sky positioning information (x, y) is contained in a file 'positions.fits'; we must combine the information in these two tables to build a (x, y, γ) table

```
shear_catalog = ShearCatalog.read('WLshear.fits')
positions_catalog = ShearCatalog.read('positions.fits')
full_catalog = tbl.hstack((positions_catalog,shear_catalog))
```

You can tweak the names of the columns that contain the position information (which by default are 'x','y') and the measure units of the positions (which by default are degrees) using the `setSpatialInfo()` method. At this point you can construct a shear grid out of the catalog, specifying a grid size and a smoothing scale.

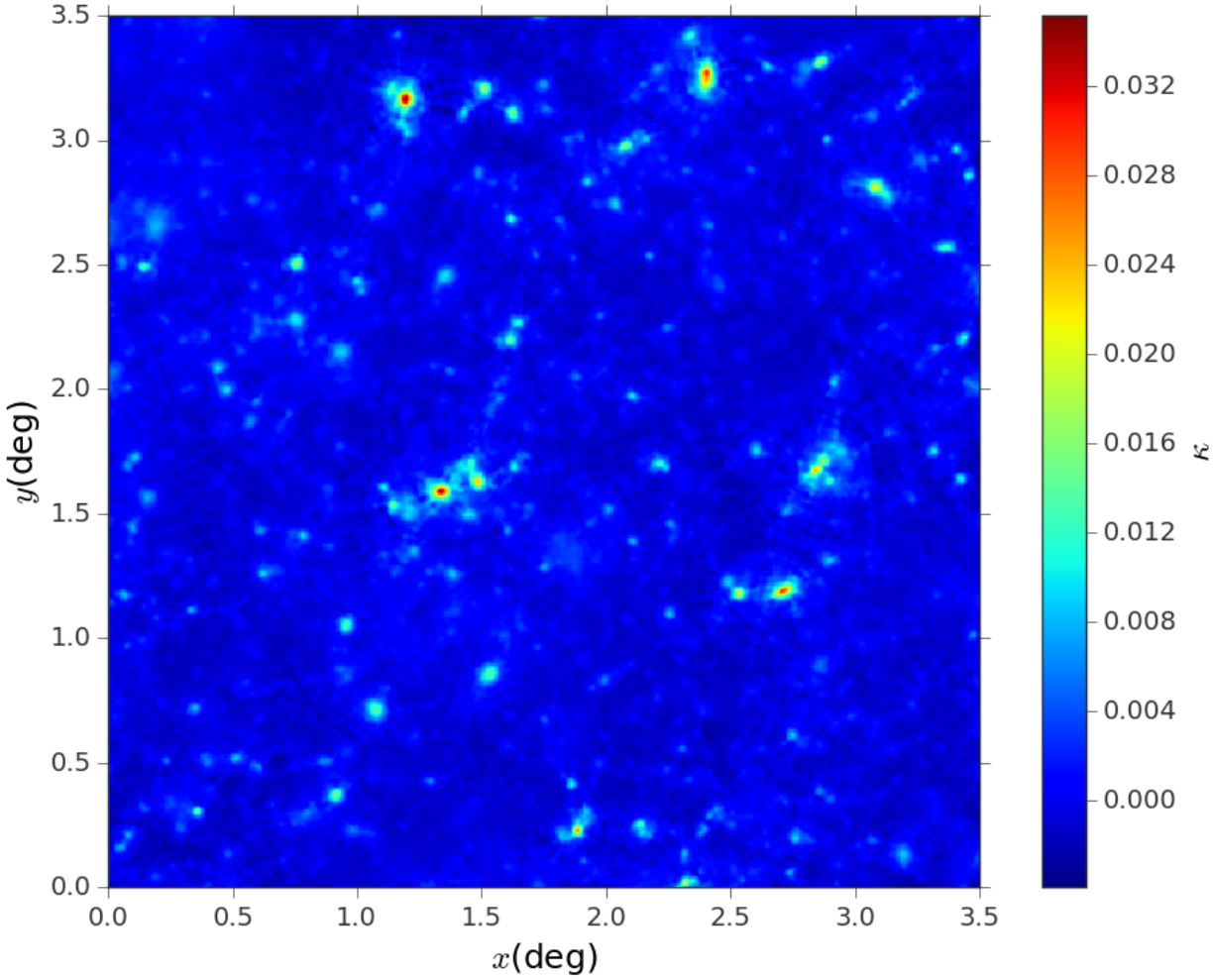
```
shear_map = full_catalog.toMap(map_size=3.5*u.deg,npixel=512,smooth=0.5*u.arcmin)
convergence_map = shear_map.convergence()
```

Note that 'shear_map' now is a *ShearMap* instance and 'convergence_map' is a *ConvergenceMap* instance, and as such you can visualize it with the *visualize()* method

```
convergence_map.visualize(colorbar=True,cbar_label=r"$\kappa$")
```

7.1.7 Decompose a shear map into E and B modes

```
from lenstools import ShearMap
from lenstools.utils.defaults import load_fits_default_convergence,load_fits_default_shear

import numpy as np
import matplotlib.pyplot as plt

from astropy.io import fits
from astropy.units import deg

def two_file_loader(filename1,filename2):

        shear_file_1 = fits.open(filename1)
        angle = shear_file_1[0].header["ANGLE"]
        gamma = shear_file_1[0].data.astype(np.float)
        shear_file_1.close()

        shear_file_2 = fits.open(filename2)
        assert shear_file_2[0].header["ANGLE"] == angle
        gamma = np.array((gamma,shear_file_2[0].data.astype(np.float)))
        shear_file_2.close()

        return angle*deg,gamma



test_map = ShearMap.load("Data/shear1.fit",filename2="Data/shear2.fit",format=two_file_loader)
```


3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
y(deg)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
x(deg)
0.032
0.028
0.024
0.020
0.016
0.012
0.008
0.004
0.000
κ

```
l_edges = np.arange(200.0,50000.0,200.0)

l,EE,BB,EB = test_map.decompose(l_edges,keep_fourier=True)

assert l.shape == EE.shape == BB.shape == EB.shape

fig,ax = plt.subplots()
ax.plot(l,l*(l+1)*EE/(2.0*np.pi),label=r"$P_{EE}$")
ax.plot(l,l*(l+1)*BB/(2.0*np.pi),label=r"$P_{BB}$")
ax.plot(l,l*(l+1)*np.abs(EB)/(2.0*np.pi),label=r"$\vert P_{EB}\vert$")

ax.set_xscale("log")
ax.set_yscale("log")
ax.set_xlabel(r"$l$")
ax.set_ylabel(r"$l(l+1)P_l/2\pi$")

ax.legend(loc="upper left")

plt.savefig("EB.png")
plt.clf()

fig,ax = plt.subplots()
ax.plot(l,np.abs(EB)/np.sqrt(EE*BB))
ax.set_xlabel(r"$l$")
ax.set_ylabel(r"$P_{EB}/\sqrt{P_{EE}P_{BB}}$")

plt.savefig("EB_corr.png")
```

These are the $P_{EE}(l), P_{BB}(l), P_{EB}(l)$ power spectra

and this is their cross correlation

7.1.8 Design your own simulation set

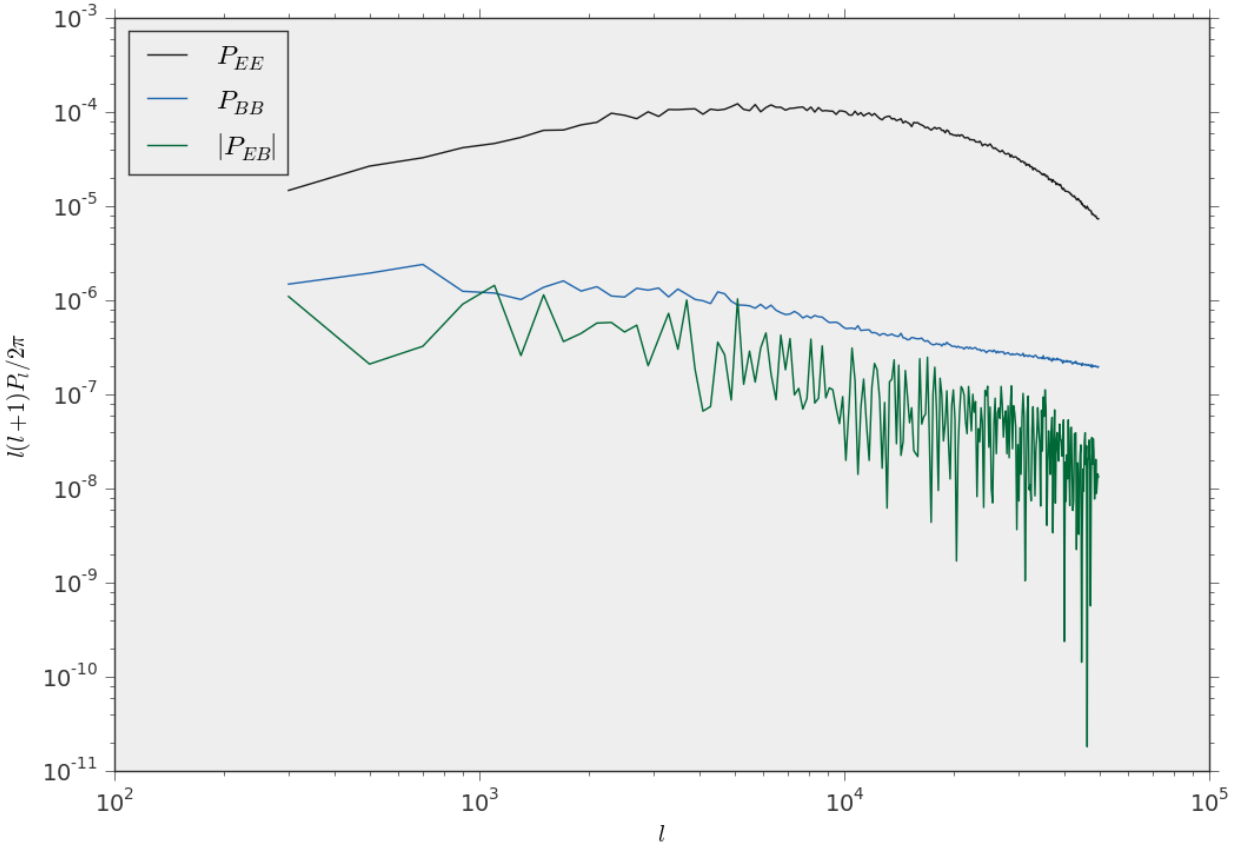
The target of this example is to get an even sampling of the cosmological parameter space in which points are as spread as possible, within some range; our first application is getting an even sampling of the (Ω_m, w, σ_8) parameter space. To do this we use a similar procedure as the Coyote2 people and reduce ourselves to the following problem: we want to draw N sample points $\mathbf{x}$ in a D dimensional hypercube of unit side ($\mathbf{x} \in [0,1]^D$) so that the points are as spread as possible. We also want to enforce the *latin* hypercube structure: when projecting the sample on each dimension, the projected points must not overlap. To solve this problem we adopt an optimization approach, in which we define a measure of how "spread" the points are; given a set of N points (or a *design* to use the same terminology as Coyote2) $\mathcal{D}$, one can define a *cost* function $d_{(p,\lambda)}(\mathcal{D})$

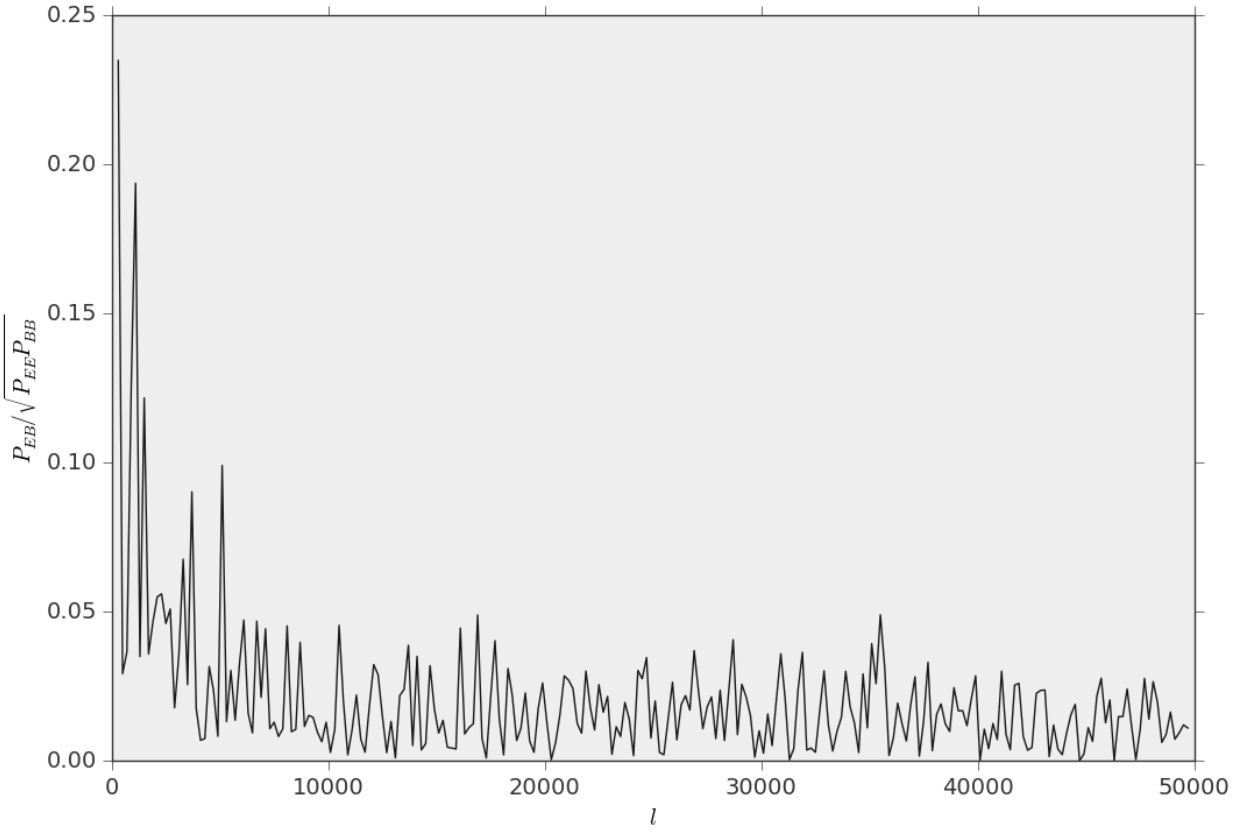
$$d_{(p,\lambda)}(\mathcal{D}) = \left(\frac{2}{N(N-1)} \sum_{i<j} \left[\frac{D^{1/p}}{\rho_p(\mathbf{x}_i, \mathbf{x}_j)} \right]^{\lambda} \right)^{1/\lambda}$$

and ρ_p is just the p-distance between two points

$$\rho_p(\mathbf{x}_i, \mathbf{x}_j) = \left[\sum_{d=1}^{D} |x_i^d - x_j^d|^p \right]^{1/p}$$

What we want is find a design $\mathcal{D}$ that minimizes the cost function maintaining the latin hypercube structure. To get an intuition of what's the meaning of this, for $(D, p, \lambda) = (3, 2, 1)$ this problem is equivalent to the one of minimizing the potential energy of N unit point charges confined in a cube: the repulsive Coulomb force will make sure that these charges are as spread apart as possible.


P_{EE}
P_{BB}
$|P_{EB}|$
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{-11}
$l(l+1)P_l/2\pi$
10^2
10^3
10^4
10^5
l



The algorithm

We use a rough approximation of the simulated annealing algorithm to perform this task: remember that we want to find a latin hypercube design that minimizes the cost function; the simplest latin hypercube one can think about is a completely diagonal one, call it $\mathcal{D}_0$, defined by the points $x_i^d \equiv i/(N-1)$, $i = 0...N-1$, for which one can compute the cost function (which will not depend on p, try to believe)

$$d_0(N,\lambda) = \left(\frac{2(N-1)^{\lambda-1}}{N} \sum_{i<j} \frac{1}{|i-j|^\lambda} \right)^{1/\lambda}$$

This design of course is far from optimal, but we can improve it by shuffling the particles coordinates for each dimension independently, this will greatly improve the cost of the design. What we can do next is exchange the single coordinates of particle pairs and see if this leads to a cost improvement or not: we can iterate this procedure many times to fine tune to the optimum. In detail the algorithm that we use consists in the following steps:

1. Start from the diagonal design $\mathcal{D}_0 : x_i^d \equiv i/(N-1)$
2. Shuffle the coordinates of the particles in each dimension independently $x_i^d = \mathcal{P}_d\left(\frac{1}{N-1}, \frac{2}{N-1}, ..., 1\right)$ where $\mathcal{P}_1, ..., \mathcal{P}_D$ are random independent permutations of $(1, 2, ..., N)$
3. Pick a random particle pair (i, j) and a random coordinate $d \in \{1, ..., D\}$ and swap $x_i^d \leftrightarrow x_j^d$
4. Compute the new cost function, if this is less than the previous step, keep the exchange, otherwise revert the coordinate swap
5. Repeat steps 3 and 4

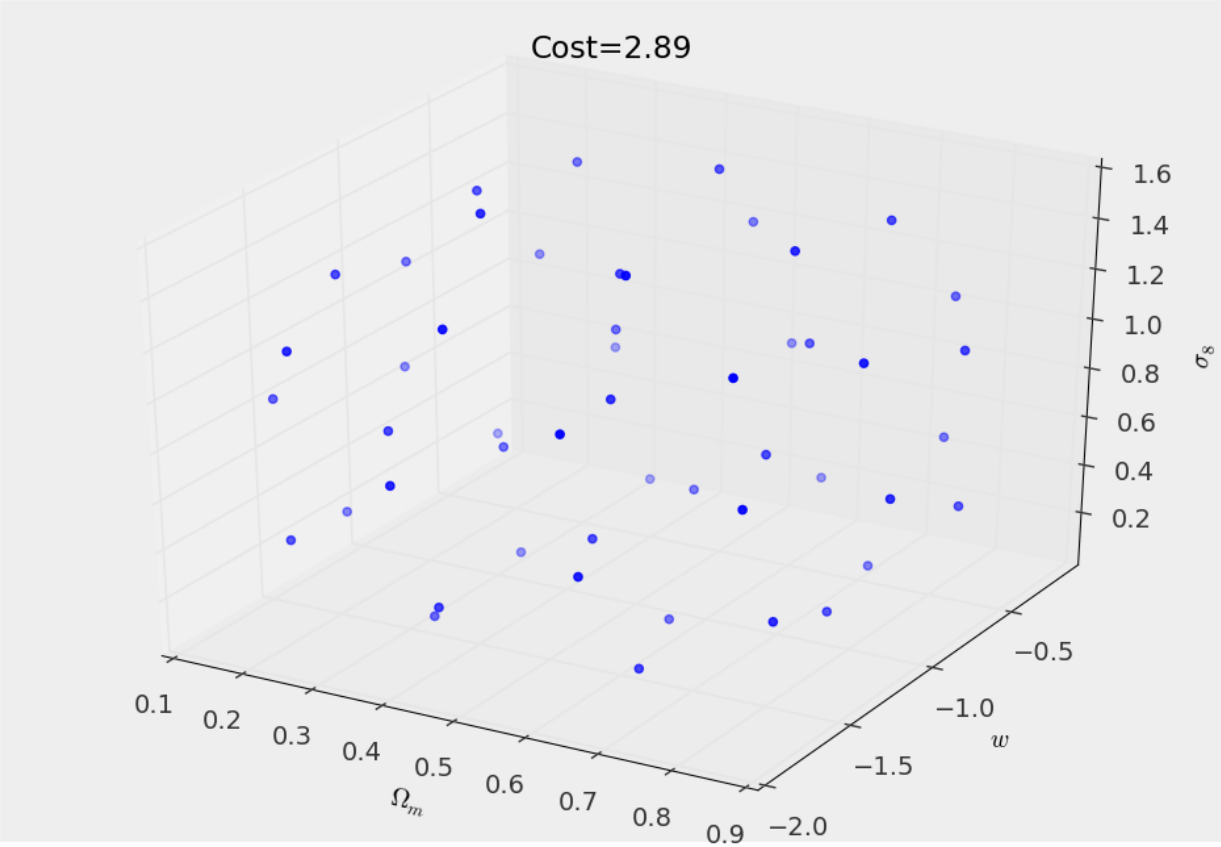
We found that 10^5 iterations are enough to achieve convergence (to at least a local optimum) for $N = 100$ points in $D = 3$ dimensions with the "Coulomb" cost function with $p = 2$ and $\lambda = 1$.

Examples

Here it is a simple example on how can you build your own simulation design using lenstools

```
from __future__ import print_function

import sys

from lenstools.simulations import Design

import numpy as np
import matplotlib.pyplot as plt

#This fails if GSL is not installed
try:
        design = Design.from_specs(npoints=50,parameters=[("Om",r"$\Omega_m$",0.1,0.9),("w",r"$w$",-2
except ImportError:
        sys.exit(1)

print(design)

#The cost function should be the diagonal one
np.testing.assert_approx_equal(design.diagonalCost(Lambda=1.0),design.cost(p=2.0,Lambda=1.0))

#Visualize the 3d diagonal configuration
design.visualize(color="blue")
design.set_title("Cost={0:.2f}".format(design.diagonalCost(Lambda=1.0)))
design.savefig("design_diagonal_3d.png")

#Visualize the 2d (Om,si8) projection
fig,ax = plt.subplots()
design.visualize(fig=fig,ax=ax,parameters=["Om","si8"],color="red",marker=".")
design.savefig("design_diagonal_2d.png")

#Now perform the sampling of the parameter space, by minimizing the cost function
deltaPerc = design.sample(Lambda=1.0,p=2.0,seed=1,maxIterations=100000)

#Visualize the 3d configuration
design.visualize(color="blue")
design.set_title("Cost={0:.2f}".format(design.cost(Lambda=1.0,p=2.0)))
design.savefig("design_3d.png")

#Visualize the 2d (Om,si8) projection
fig,ax = plt.subplots()
design.visualize(fig=fig,ax=ax,parameters=["Om","si8"],color="red",marker=".")
design.savefig("design_2d.png")

#Visualize the changes in the cost function
fig,ax = plt.subplots()
ax.plot(design.cost_values)
ax.set_xlabel(r"$N$")
ax.set_ylabel("cost")
ax.set_title("Last change={0:.1e}%".format(deltaPerc*100))
ax.set_xscale("log")
fig.savefig("cost.png")
```

This is how your design looks like in 3D space and on the (Ω_m, σ_8) projection



And this is the evolution of the cost function

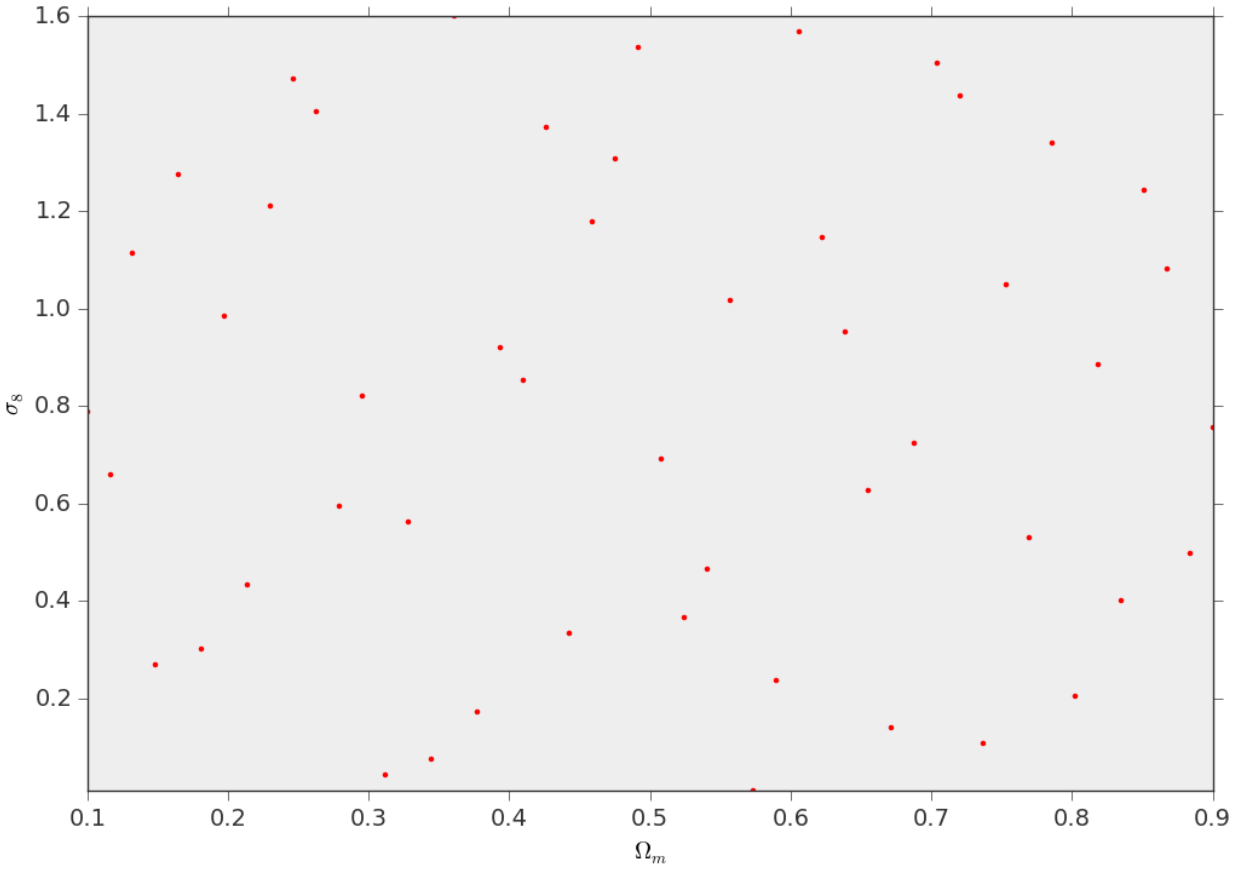
7.1.9 I/O to and from Gadget2 snapshots

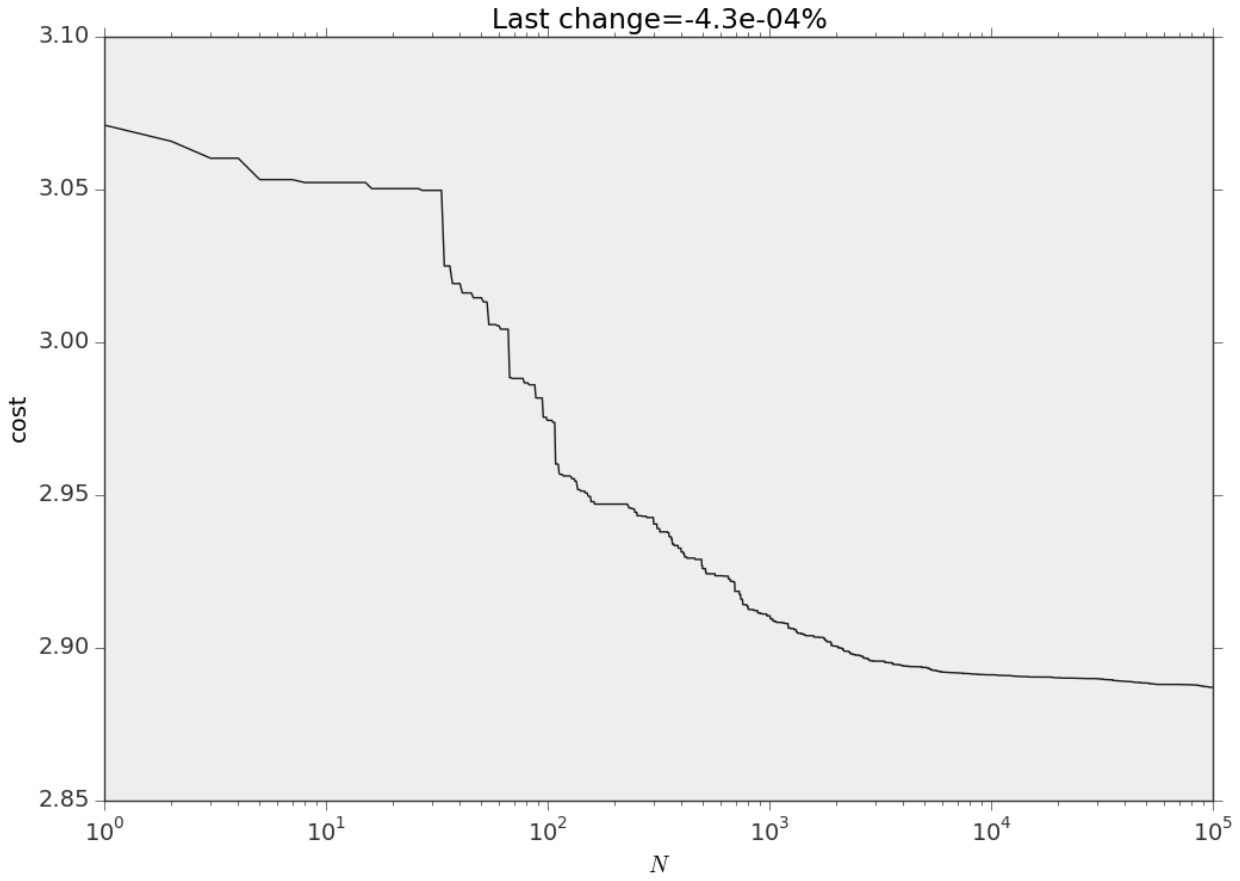
LensTools provides an easy to use API to interact with the Gadget2 binary format (complete documentation in API); you can use the numpy functionality to generate your own position and velocity fields, and then use the LensTools API to write them to a properly formatted Gadget2 snapshot (that you can subsequently evolve with Gadget2). Here's an example with 32^3 particles distributed normally around the center of the box (15 Mpc), with uniform velocities in [-1,1]m/s. First we generate the profiles

```
>>> from lenstools.simulations import Gadget2SnapshotDE

>>> import numpy as np
>>> from astropy.units import Mpc,m,s

#Generate random positions and velocities
>>> NumPart = 32**3
>>> x = np.random.normal(loc=7.0,scale=5.0,size=(NumPart,3)) * Mpc
>>> v = np.random.uniform(-1,1,size=(NumPart,3)) * m / s
```

Then we write them to a snapshot

```
#########################Write###################################
```


1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
σ_8
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Ω_m


Last change=-4.3e-04%
3.10
3.05
3.00
2.95
2.90
2.85
cost
10^0
10^1
10^2
10^3
10^4
10^5
N

```
#Create an empty gadget snapshot

>>> snap = Gadget2SnapshotDE()

#Put the particles in the snapshot
>>> snap.setPositions(x)
>>> snap.setVelocities(v)

#Generate minimal header
>>> snap.setHeaderInfo()

#Write the snapshot
>>> snap.write("gadget_ic")
```

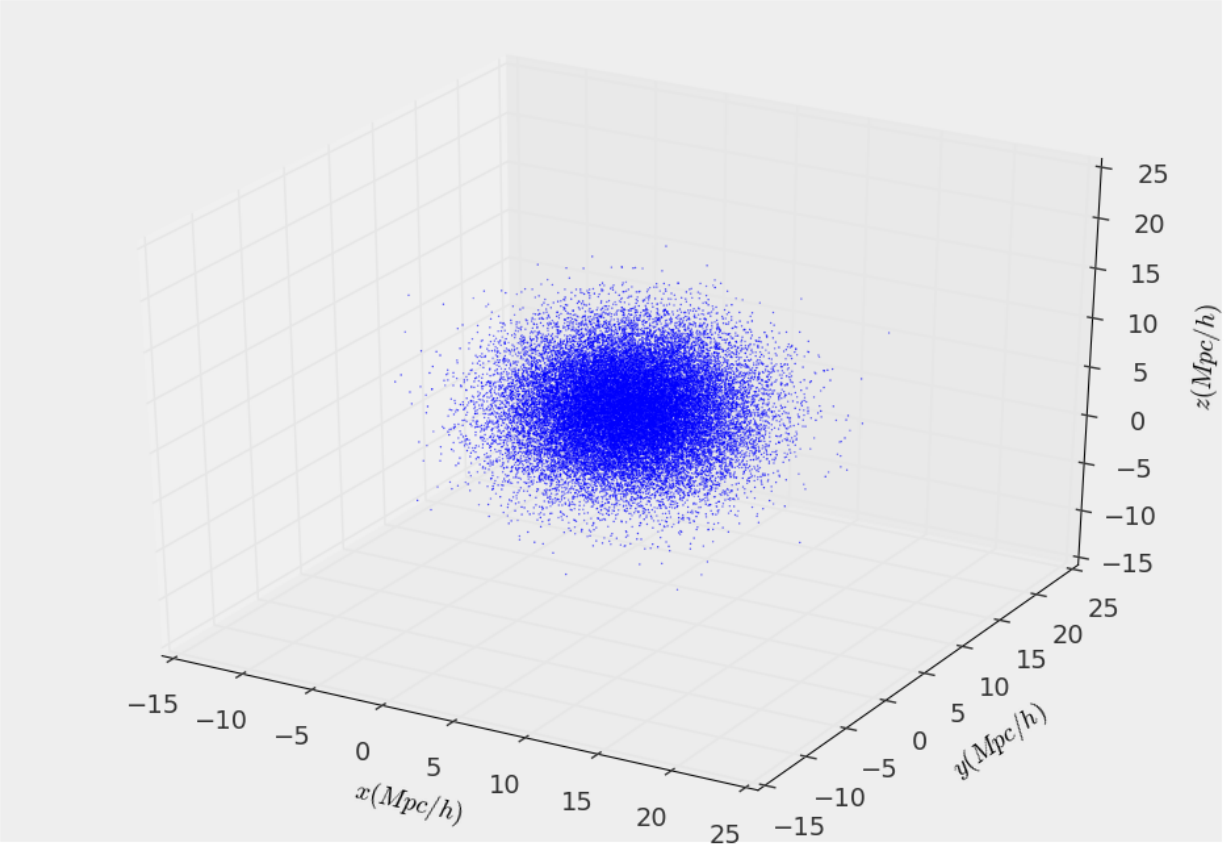
Now check that we did everything correctly, visualizing the snapshot

```
######################Read and visualize############################

#Open the snapshot
>>> snap = Gadget2SnapshotDE.open("gadget_ic")

#Visualize the header
>>> print(snap.header)

#Get positions and velocities
>>> snap.getPositions()
>>> snap.getVelocities()

#Visualize the snapshot
>>> snap.visualize(s=1)
>>> snap.savefig("snapshot.png")
>>> snap.close()
```

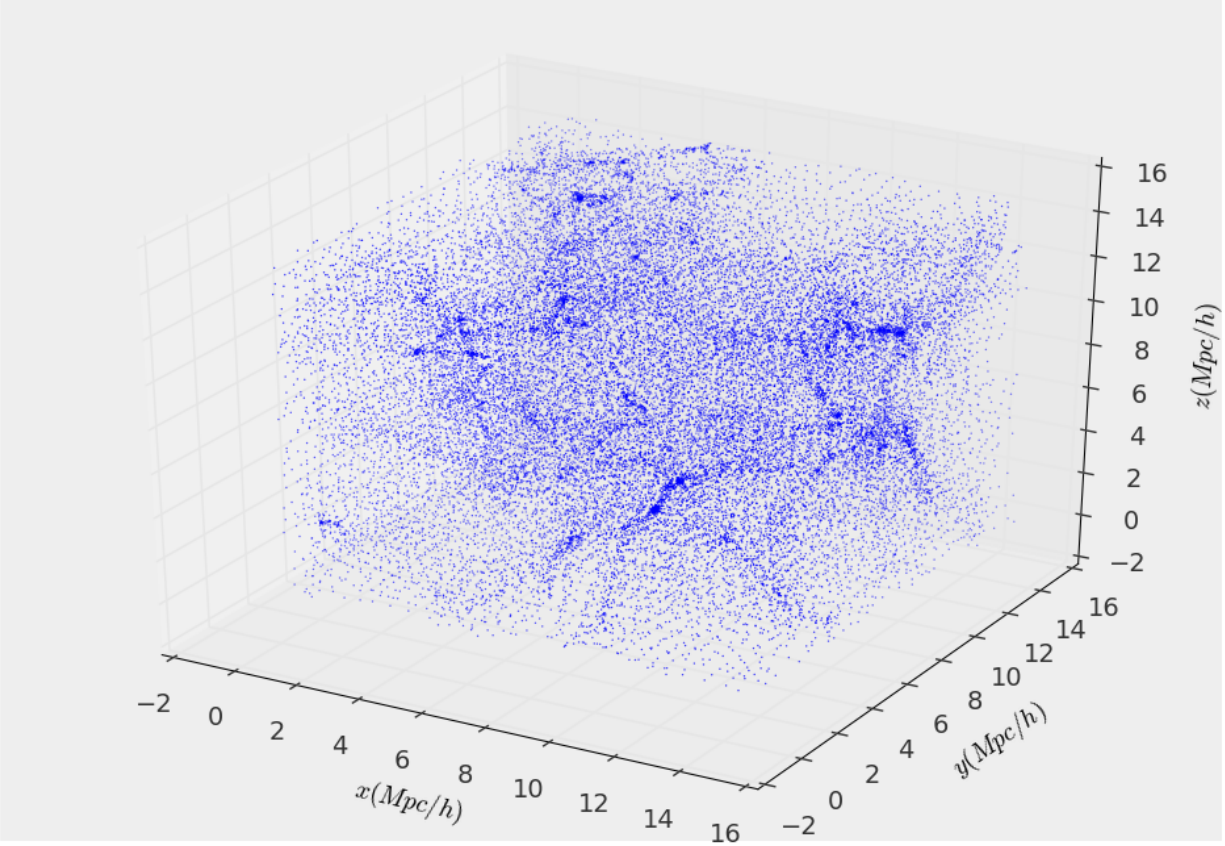
This is the result

If you don't believe that this works, here it is what happens with an actual snapshot produced by a run of Gadget2

```
######################Read and visualize############################

#Open the snapshot
>>> snap = Gadget2SnapshotDE.open("../Test/Data/gadget/snapshot_001")

#Visualize the header
>>> print(snap.header)

H0 : 72.0 km / (Mpc s)
Ode0 : 0.74
Om0 : 0.26
box_size : 15.0 Mpc/h
endianness : 0
files : ['Test/Data/gadget/snapshot_001']
flag_cooling : 0
flag_feedback : 0
flag_sfr : 0
h : 0.72
masses : [  0.00000000e+00   1.03224800e+10   0.00000000e+00   0.00000000e+00 0.00000000e+00   0.0000
num_files : 1
num_particles_file : 32768
num_particles_file_gas : 0
```


25
20
15
10
5
0
−5
−10
−15
$z(Mpc/h)$
25
20
15
10
5
0
−5
−10
−15
$y(Mpc/h)$
−15
−10
−5
0
5
10
15
20
25
$x(Mpc/h)$

```
num_particles_file_of_type : [     0 32768      0      0      0      0]
num_particles_file_with_mass : 0
num_particles_total : 32768
num_particles_total_gas : 0
num_particles_total_of_type : [     0 32768      0      0      0      0]
num_particles_total_side : 32
num_particles_total_with_mass : 0
redshift : 2.94758939237
scale_factor : 0.253319152679
w0 : -1.0
wa : 0.0

#Get positions and velocities
snap.getPositions()
snap.getVelocities()

#Visualize the snapshot
snap.visualize(s=1)
snap.savefig("snapshot_gadget.png")
```

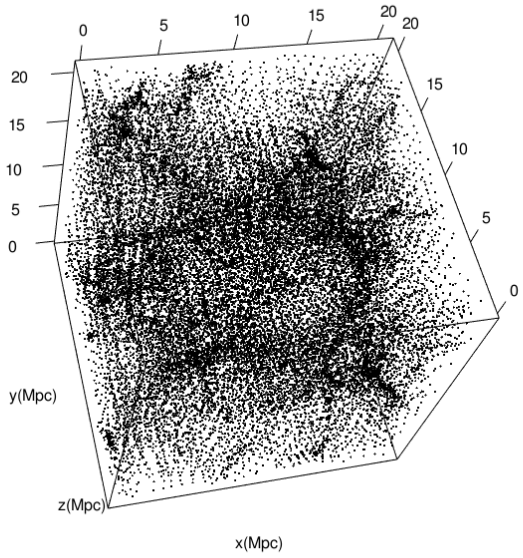


If you wish, you can export the snapshot positions in R format so that you can take full advantage of the RGL graphics library to visualize your snapshot (works a lot better than matplotlib for three dimensional plots):

```
#Save positions in R format
snap.pos2R("snapshot.rdata",variable_name="pos")

################################################
```

```
#####Then, inside an R console###################
################################################

library('rgl')
load('snapshot.rdata')
n <- 32^3
plot3d(pos[1:n,1],pos[1:n,2],pos[1:n,3],size=1,xlab='x(Mpc)',ylab='y(Mpc)',zlab='z(Mpc)')
rgl.snapshot( 'snapshot_R.png', fmt = "png", top = TRUE )
```

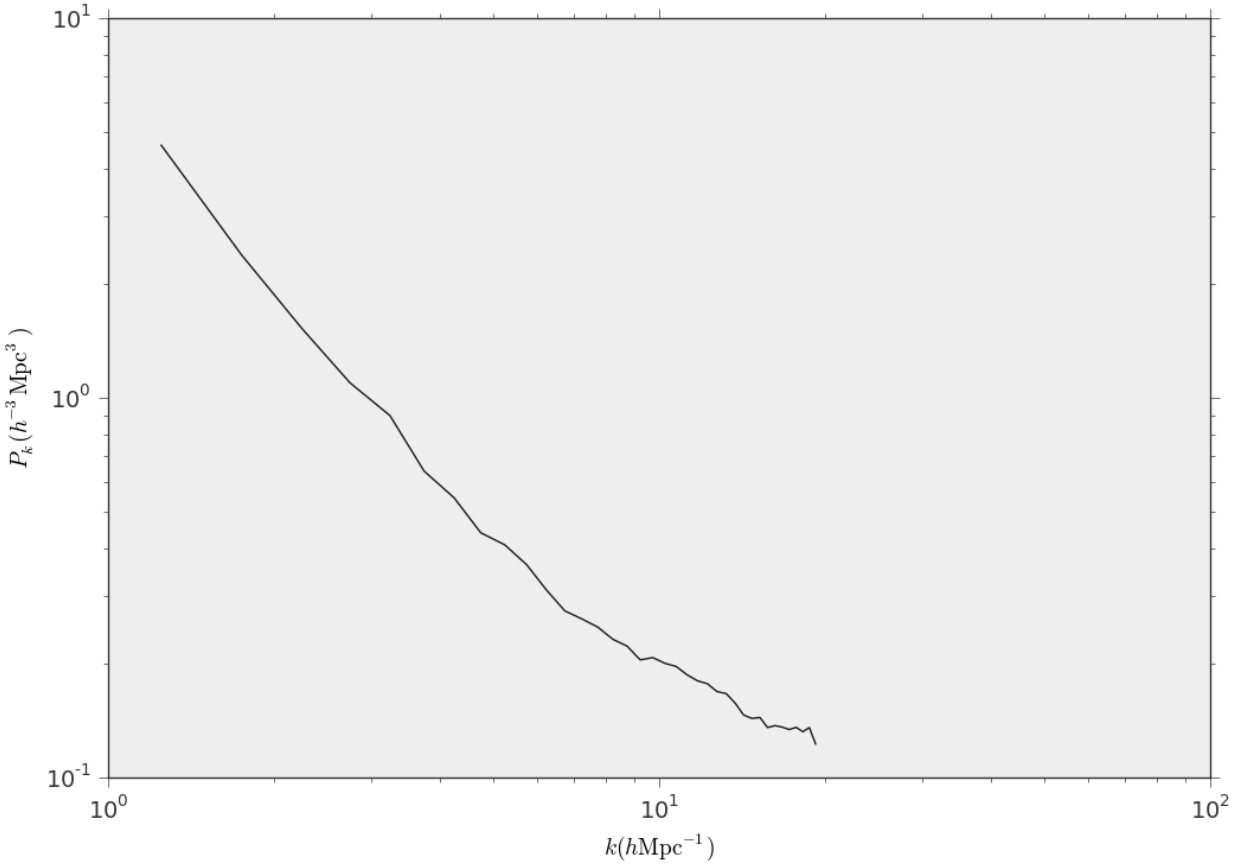
which looks something like this



We can also measure the density fluctuations power spectrum P_k, defined as $\langle \delta n_k \delta n_{k'} \rangle = \delta_D(k+k')P_k$

```
#Measure the power spectrum
k_edges = np.arange(1.0,20.0,0.5) * (1/Mpc)
k,Pk = snap.powerSpectrum(k_edges,resolution=64)

#Plot
fig,ax = plt.subplots()

ax.plot(k,Pk)
ax.set_yscale("log")
ax.set_xscale("log")
ax.set_xlabel(r"$k(h\mathrm{Mpc}^{-1})$")
ax.set_ylabel(r"h^{-3}$P_k(\mathrm{Mpc}^3)$")
fig.savefig("snapshot_power_spectrum.png")
snap.close()
```

Which looks like this


10^1
10^0
10^{-1}
$P_k\,(h^{-3}\mathrm{Mpc}^3\,)$
10^0
10^1
10^2
$k(h\mathrm{Mpc}^{-1})$

7.1.10 Constraining cosmological (and not) parameter spaces

This brief tutorial shows how to use lenstools to use the emulator and contour plotting features to produce publication quality confidence contour plots. This example has nothing to do with cosmology or weak lensing, but the methods are immediately transferrable to these fields. The tutorial is also available in this notebook.

First we need to set up the environment

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from lenstools.statistics.ensemble import Ensemble,Series
from lenstools.statistics.constraints import Emulator
from lenstools.statistics.contours import ContourPlot
```

The next step is to generate some fake training data (in a weak lensing analysis these will be provided by the simulations: the parameters will be the cosmological parameters, the features will be power spectra, etc...)

```
p = np.outer(np.arange(10.0),np.ones(2))
f = np.outer(np.arange(10.0)+0.1*2,np.ones(5))
emulator = Emulator.from_features(f,p,parameter_index=["alpha","beta"],feature_index=[r"$f_{0}$".form
```

Next we generate some fake test feature to fit (in a weak lensing context these will be the actual data)

```
test_feature = Series(np.ones(5)*4.0 + 0.05*np.random.randn(5),index=emulator[["features"]].columns)
```

We need to assume a particular form for the feature covariance matrix: the simplest one is a diagonal form

```
features_covariance = Ensemble(np.eye(5)*0.5,index=test_feature.index,columns=test_feature.index)
```

Next we train the emulator (basically set it up to interpolate the features between the training points provided)

```
emulator.train()
```

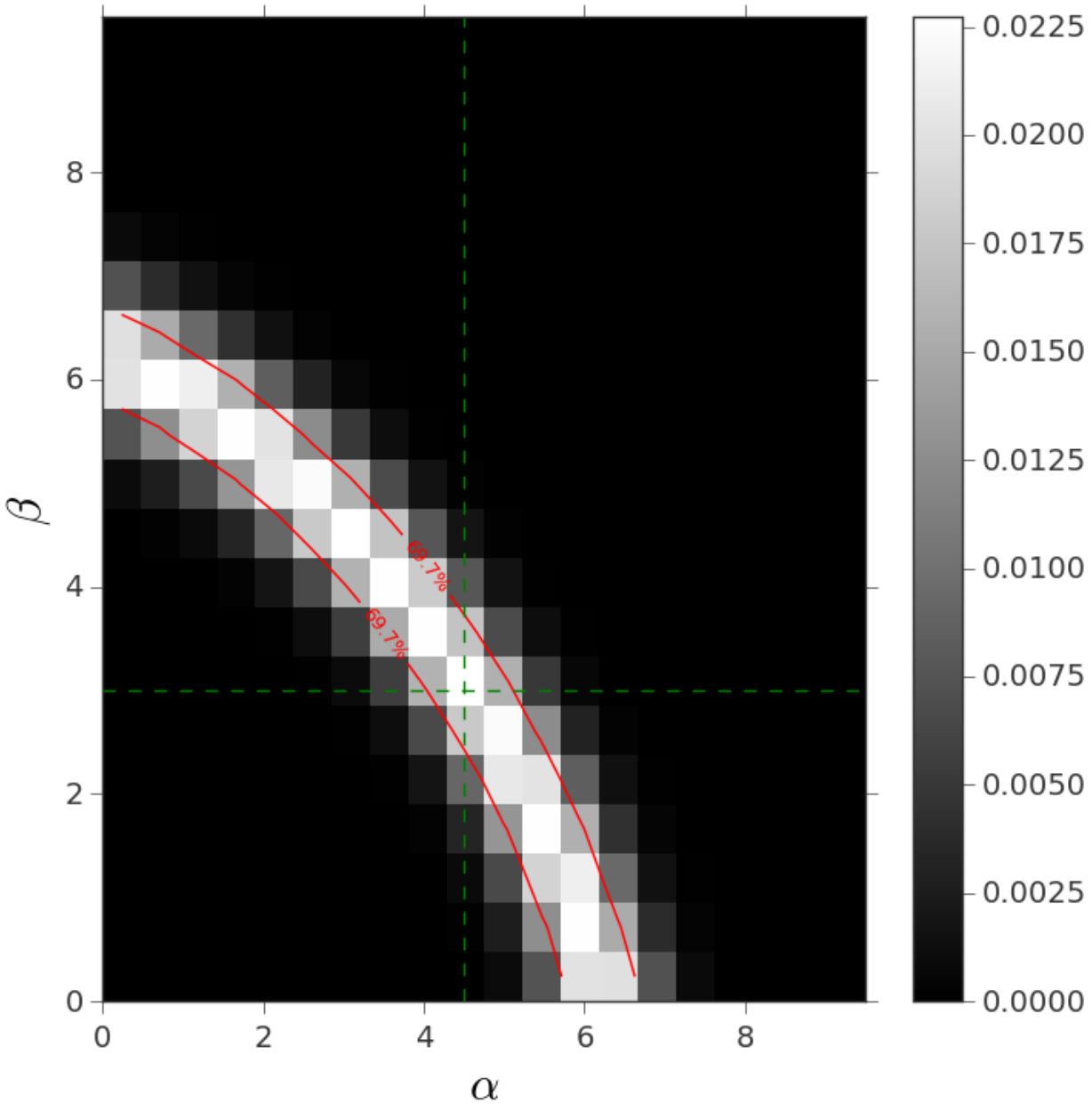
We then build a grid of possible parameter combinations to assign to the test_feature, and we evaluate the score of each of these parameter combinations

```
#Grid of parameters
g = np.arange(0.0,10.0,0.5)
p = np.array(np.meshgrid(g,g,indexing="ij")).reshape(2,400).T
test_parameters = Ensemble(p,columns=emulator.parameter_names)

#Scores are chi squared: the probability is exp(-0.5*chi2)
scores = emulator.score(parameters=test_parameters,observed_feature=test_feature,features_covariance=
scores['features'] = np.exp(-0.5*scores['features'])
```

Finally we plot the (alpha,beta) confidence contours

```
contour = ContourPlot.from_scores(scores,parameters=emulator.parameter_names,plot_labels=[r"$\alpha$"
contour.show()
contour.getLikelihoodValues([0.684],precision=0.1)
contour.plotContours(colors=["red"])
contour.labels()

contour.savefig("contour_example.png")
```

This is how the result looks like


0.0225
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
8
6
4
2
0
0
2
4
6
8
β
α
69.7%
69.7%

CHAPTER 8

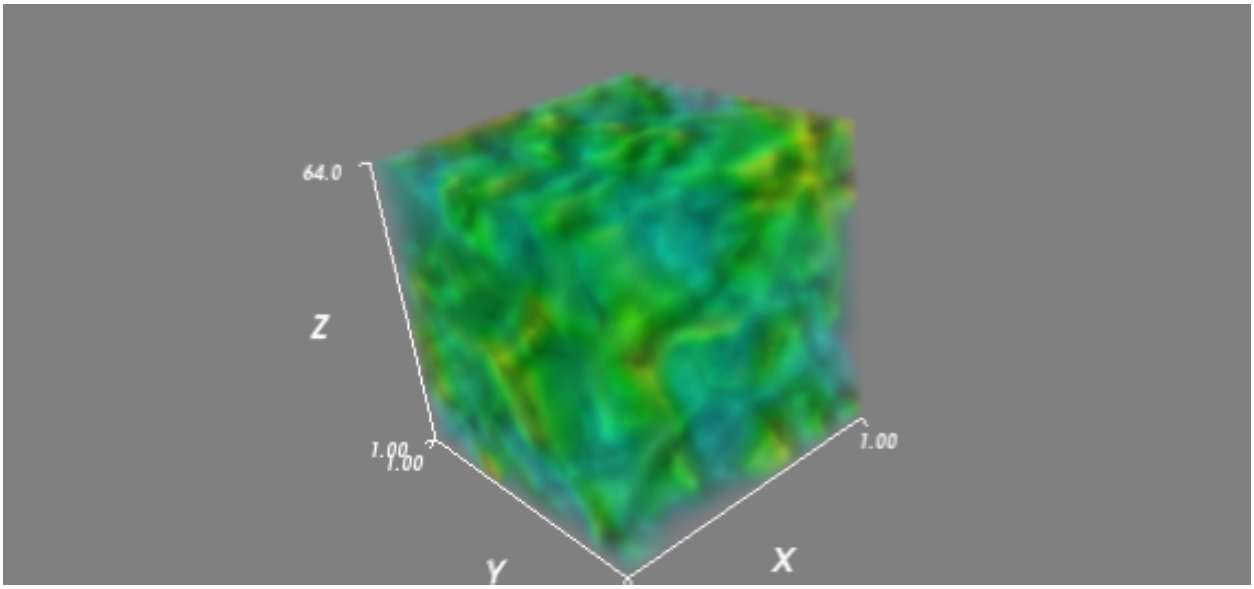
3D visualization with Mayavi

8.1 Visualization

This is a gallery of figures produced with the 3D visualization engine Mayavi

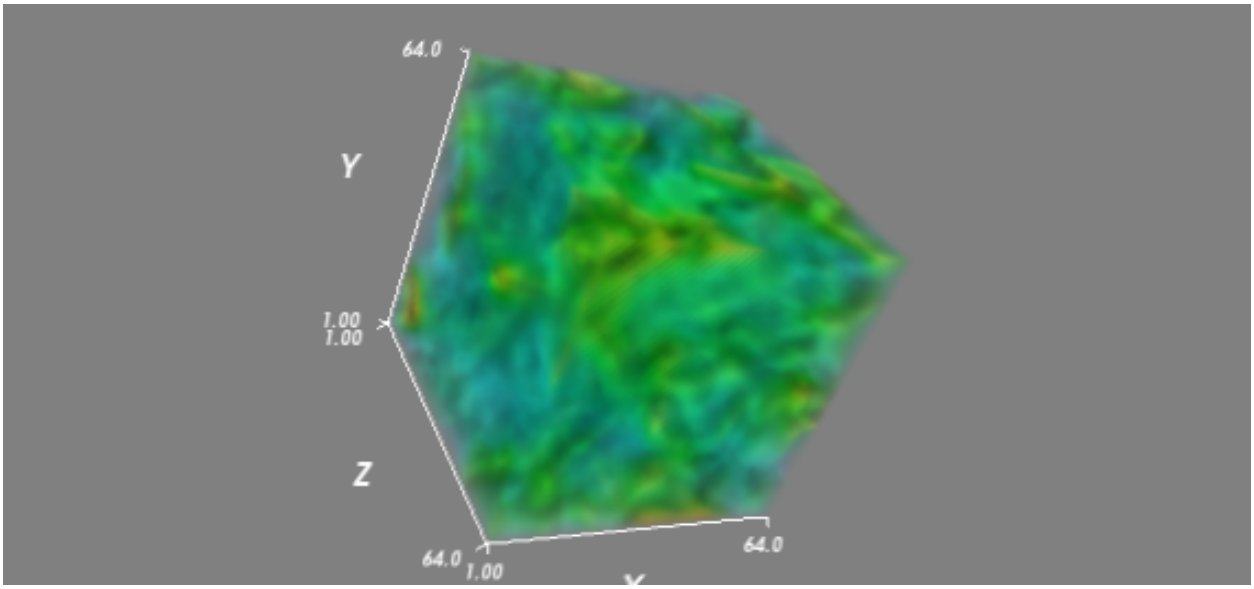
8.1.1 Three dimensional density field visualization

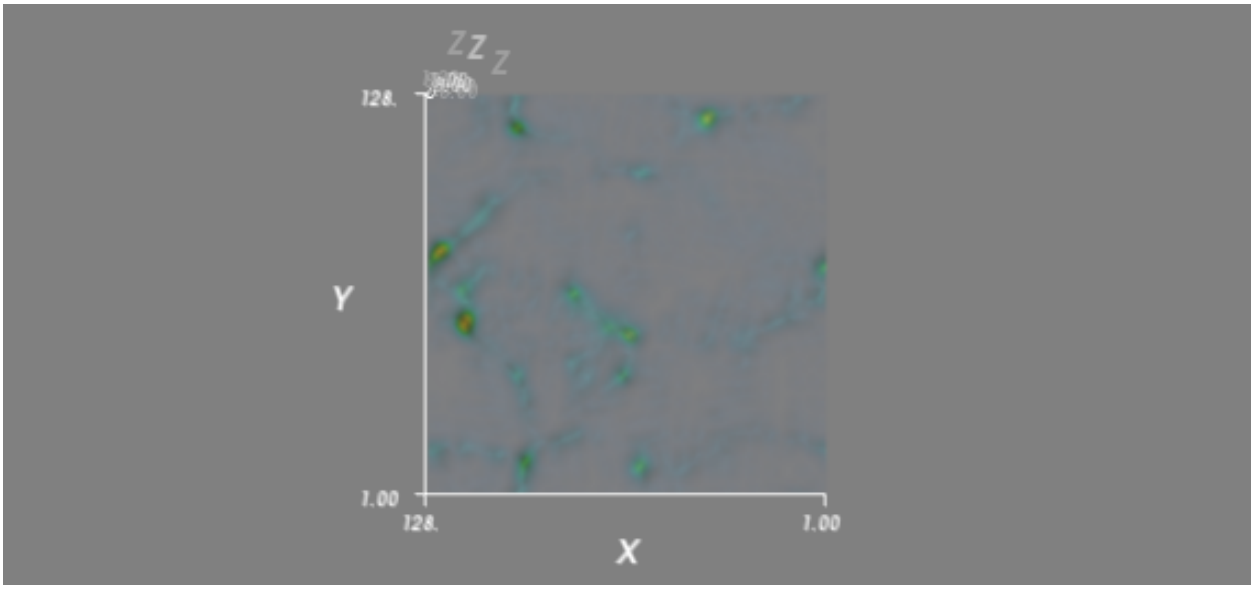
This is how a Gadget snapshot looks like

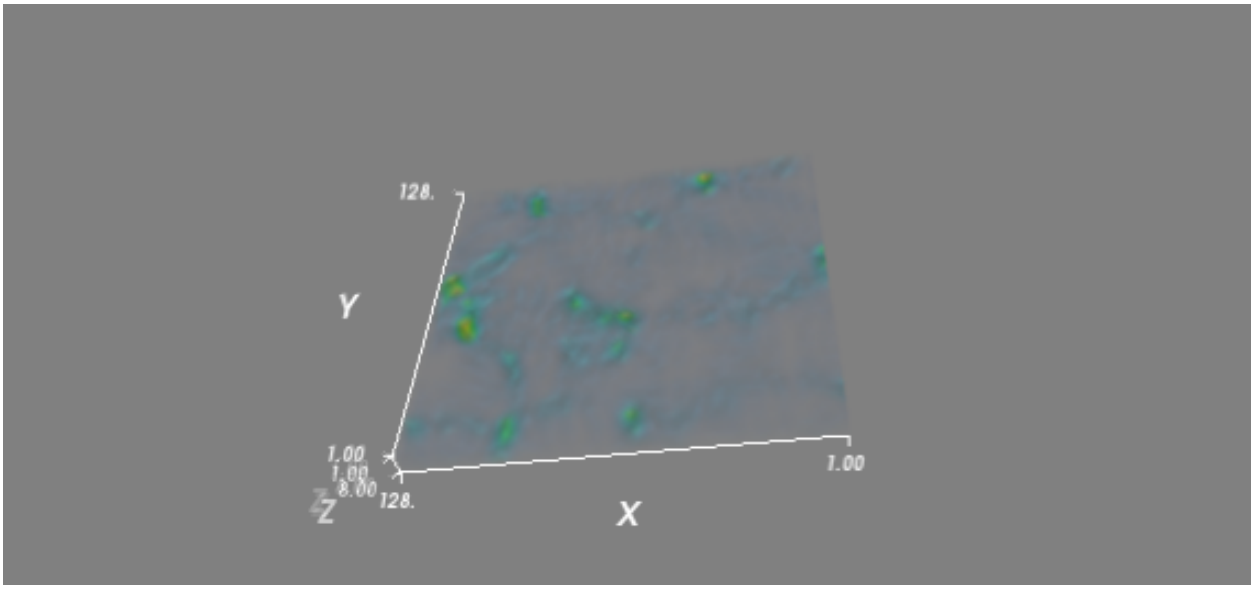


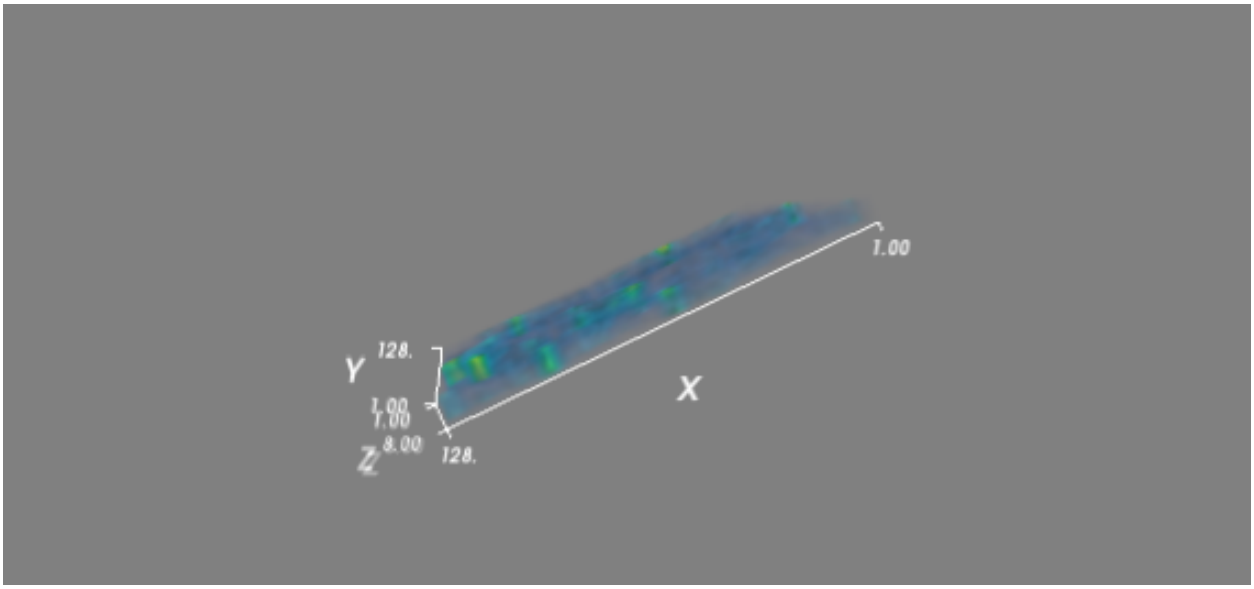
8.1.2 Lens plane visualization

And this is how a lens plane looks like


64.0
Y
1.00
1.00
Z
64.0
1.00
64.0


Z
128.
Y
1.00
128.
1.00
X


128.
Y
1.00
1.00
8.00
128.
Z
X
1.00


1.00
128.
Y
1.00
1.00
8.00
128.
Z
X

CHAPTER 9

API

9.1 API

A collection of tools widely used in Weak Gravitational Lensing data analyses

9.1.1 Convergence maps

class lenstools.image.convergence.**ConvergenceMap**(*data*, *angle*, *masked=False*, ***kwargs*)

A class that handles 2D convergence maps and allows to compute their topological descriptors (power spectrum, peak counts, minkowski functionals)

```
>>> from lenstools import ConvergenceMap
>>> from lenstools.defaults import load_fits_default_convergence
```

```
>>> test_map = ConvergenceMap.load("map.fit",format=load_fits_default_convergence)
>>> imshow(test_map.data)
```

boundary

Computes the boundaries of the masked regions, defined as the regions in which the convergence is still well defined but the first and second derivatives are not

Returns array of bool. of the same shape as the map, with True values along the boundaries

countModes(*l_edges*)

Counts the available number of modes in multipole space available to each bin specified in l_edges

Parameters **l_edges** (*array*) – Multipole bin edges

Returns number of modes available to each bin

Return type array

Raises AssertionError if l_edges are not provided

cross(*other*, *statistic='power_spectrum'*, ***kwargs*)

Measures a cross statistic between maps

Parameters

- **other** (*ConvergenceMap instance*) – The other convergence map
- **statistic** (*string or callable*) – the cross statistic to measure (default is 'power_spectrum')
- **kwargs** (*dict.*) – the keyword arguments are passed to the statistic (when callable)

Returns tuple – (l – array,Pl – array) = (multipole moments, cross power spectrum at multipole moments) if the statistic is the power spectrum, otherwise whatever statistic() returns on call

Raises AssertionError if the other map has not the same shape as the input one

```
>>> test_map = ConvergenceMap.load("map.fit",format=load_fits_default_convergence)
>>> other_map = ConvergenceMap.load("map2.fit",format=load_fits_default_convergence)
```

```
>>> l_edges = np.arange(200.0,5000.0,200.0)
>>> l,Pl = test_map.cross(other_map,l_edges=l_edges)
```

cutRegion(*extent*)

Cut a subset of the map, with the same resolution (warning! tested on square cuts only!)

Parameters **extent** (*array with units*) – region in the map to select (xmin,xmax,ymin,ymax), must have units

Returns new ConvergenceMap instance that encloses the selected region

getValues(*x*, *y*)

Extract the map values at the requested (x,y) positions; this is implemented using the numpy fast indexing routines, so the formats of x and y must follow the numpy advanced indexing rules. Periodic boundary conditions are enforced

Parameters

- **x** (*numpy array or quantity*) – x coordinates at which to extract the map values (if unitless these are interpreted as radians)
- **y** (*numpy array or quantity*) – y coordinates at which to extract the map values (if unitless these are interpreted as radians)

Returns numpy array with the map values at the specified positions, with the same shape as x and y

Raises IndexError if the formats of x and y are not the proper ones

gradient(*x=None*, *y=None*, *save=True*)

Computes the gradient of the map and sets the gradient_x,gradient_y attributes accordingly

Parameters

- **x** (*array with units*) – optional, x positions at which to evaluate the gradient
- **y** (*array with units*) – optional, y positions at which to evaluate the gradient
- **save** (*bool.*) – if True saves the gradient as attrubutes

Returns tuple – (gradient_x,gradient_y)

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> gx,gy = test_map.gradient()
```

hessian(*x=None*, *y=None*, *save=True*)

Computes the hessian of the map and sets the hessian_xx,hessian_yy,hessian_xy attributes accordingly

Parameters

- **x** (*array with units*) – optional, x positions at which to evaluate the hessian
- **y** (*array with units*) – optional, y positions at which to evaluate the hessian
- **save** (*bool.*) – if True saves the gradient as attrubutes

Returns tuple – (hessian_xx,hessian_yy,hessian_xy)

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> hxx,hyy,hxy = test_map.hessian()
```

info
Displays some of the information stored in the map (mainly resolution)

load (*filename*, *format=None*, ***kwargs*)
This class method allows to read the map from a data file, in various formats

Parameters

- **filename** (*str.*) – name of the file in which the map is saved
- **format** (*str. or callable*) – the format of the file in which the map is saved (can be a callable too); if None, it's detected automatically from the filename
- **kwargs** (*dict.*) – the keyword arguments are passed to the format (if callable)

Returns Spin0 instance with the loaded map

locatePeaks (*thresholds*, *norm=False*)
Locate the peaks in the map

Parameters

- **thresholds** (*array*) – thresholds extremes that define the binning of the peak histogram
- **norm** (*bool.*) – normalization; if set to a True, interprets the thresholds array as units of sigma (the map standard deviation)

Returns tuple – (peak height – array, peak locations – array with units)

Raises AssertionError if thresholds array is not provided

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> thresholds = np.arange(map.data.min(),map.data.max(),0.05)
>>> height,positions = test_map.locatePeaks(thresholds)
```

mask (*mask_profile*, *inplace=False*)
Applies a mask to the convergence map: all masked pixels are given a nan value because they cannot be used in the calculations

Parameters

- **mask_profile** (*array. or ConvergenceMap instance*) – profile of the mask, must be an array of 1 byte intergers that are either 0 (if the pixel is masked) or 1 (if the pixel is not masked). Must be of the same shape as the original map
- **inplace** – if True the masking is performed in place and the original map is lost, otherwise a new instance of ConvergenceMap with the masked map is returned

Returns the masked convergence map if inplace is False, otherwise a float corresponding to the masked fraction of the map

maskBoundaries ()
Computes the mask boundaries defined in the following way: a boundary is a region where the convergence value is defined, but the gradients are not defined.

Returns float. : perimeter/area ratio of the mask boundaries

maskedFraction
Computes the masked fraction of the map

Returns float, the masked fraction of the map

mean()
Computes the mean value of the pixels, taking into account eventual masking

Returns float.

minkowskiFunctionals(*thresholds*, *norm=False*)
Measures the three Minkowski functionals (area,perimeter and genus characteristic) of the specified map excursion sets

Parameters

- **thresholds** (*array*) – thresholds that define the excursion sets to consider
- **norm** (*bool.*) – normalization; if set to a True, interprets the thresholds array as units of sigma (the map standard deviation)

Returns tuple – (nu – array, V0 – array, V1 – array, V2 – array) nu are the bins midpoints and V are the Minkowski functionals

Raises AssertionError if thresholds array is not provided

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> thresholds = np.arange(-2.0,2.0,0.2)
>>> nu,V0,V1,V2 = test_map.minkowskiFunctionals(thresholds,norm=True)
```

moments(*connected=False*, *dimensionless=False*)
Measures the first nine moments of the convergence map (two quadratic, three cubic and four quartic)

Parameters

- **connected** (*bool.*) – if set to True returns only the connected part of the moments
- **dimensionless** (*bool.*) – if set to True returns the dimensionless moments, normalized by the appropriate powers of the variance

Returns array – (sigma0,sigma1,S0,S1,S2,K0,K1,K2,K3)

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> var0,var1,sk0,sk1,sk2,kur0,kur1,kur2,kur3 = test_map.moments()
>>> sk0,sk1,sk2 = test_map.moments(dimensionless=True)[2:5]
>>> kur0,kur1,kur2,kur3 = test_map.moments(connected=True,dimensionless=True)[5:]
```

pdf(*thresholds*, *norm=False*)
Computes the one point probability distribution function of the convergence map

Parameters

- **thresholds** (*array*) – thresholds extremes that define the binning of the pdf
- **norm** (*bool.*) – normalization; if set to a True, interprets the thresholds array as units of sigma (the map standard deviation)

Returns tuple – (threshold midpoints – array, pdf normalized at the midpoints – array)

Raises AssertionError if thresholds array is not provided

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> thresholds = np.arange(map.data.min(),map.data.max(),0.05)
>>> nu,p = test_map.pdf(thresholds)
```

peakCount(*thresholds*, *norm=False*)
Counts the peaks in the map

Parameters

- **thresholds** (*array*) – thresholds extremes that define the binning of the peak histogram
- **norm** (*bool.*) – normalization; if set to a True, interprets the thresholds array as units of sigma (the map standard deviation)

Returns tuple – (threshold midpoints – array, differential peak counts at the midpoints – array)

Raises AssertionError if thresholds array is not provided

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> thresholds = np.arange(map.data.min(),map.data.max(),0.05)
>>> nu,peaks = test_map.peakCount(thresholds)
```

peakHistogram(*thresholds*, *norm=False*, *fig=None*, *ax=None*, ***kwargs*)
Plot the peak histogram of the map

plotPDF(*thresholds*, *norm=False*, *fig=None*, *ax=None*, ***kwargs*)
Plot the PDF of the map

plotPowerSpectrum(*l_edges*, *show_angle=True*, *angle_unit=Unit("arcmin")*, *fig=None*, *ax=None*, ***kwargs*)
Plot the power spectrum of the map

powerSpectrum(*l_edges*)
Measures the power spectrum of the convergence map at the multipole moments specified in the input

Parameters **l_edges** (*array*) – Multipole bin edges

Returns tuple – (l – array,Pl – array) = (multipole moments, power spectrum at multipole moments)

Raises AssertionError if l_edges are not provided

```
>>> test_map = ConvergenceMap.load("map.fit")
>>> l_edges = np.arange(200.0,5000.0,200.0)
>>> l,Pl = test_map.powerSpectrum(l_edges)
```

save(*filename*, *format=None*, *double_precision=False*)
Saves the map to an external file, of which the format can be specified (only fits implemented so far)

Parameters

- **filename** (*str.*) – name of the file on which to save the plane
- **format** (*str.*) – format of the file, only FITS implemented so far; if None, it's detected automatically from the filename
- **double_precision** (*bool.*) – if True saves the Plane in double precision

savefig(*filename*)
Saves the map visualization to an external file

Parameters **filename** (*str.*) – name of the file on which to save the map

setAngularUnits(*unit*)
Convert the angular units of the map to the desired unit

Parameters **unit** (*astropy units*) – astropy unit instance to which to perform the conversion

smooth(*scale_angle*, *kind='gaussian'*, *inplace=False*, ***kwargs*)
Performs a smoothing operation on the convergence map

Parameters

- **scale_angle** (*float.*) – size of the smoothing kernel (must have units)
- **kind** (*str.*) – type of smoothing to be performed. Select "gaussian" for regular Gaussian smoothing in real space or "gaussianFFT" if you want the smoothing to be performed via FFTs (advised for large scale_angle)
- **inplace** (*bool.*) – if set to True performs the smoothing in place overwriting the old convergence map
- **kwargs** (*dict.*) – the keyword arguments are passed to the filter function

Returns ConvergenceMap instance (or None if inplace is True)

std()
Computes the standard deviation of the pixels, taking into account eventual masking

Returns float.

twoPointFunction(*algorithm='FFT'*, ***kwargs*)
Computes the two point function of the convergence

Parameters

- **algorithm** (*str.*) – algorithm used to measure the two point function. Can be "FFT"
- **kwargs** (*dict.*) – accepted keyword arguments are "theta" to specify the angles at which to measure the 2pcf. If none indicated, the angles are computed automatically

Returns (theta,2pcf(theta))

Return type tuple.

visualize(*fig=None*, *ax=None*, *colorbar=False*, *cmap='jet'*, *cbar_label=None*, ***kwargs*)
Visualize the convergence map; the kwargs are passed to imshow

class lenstools.image.convergence.**Mask**(*data*, *angle*, *masked=False*)
A class that handles the convergence masked regions

9.1.2 Shear maps and catalogs

class lenstools.image.shear.**ShearMap**(*data*, *angle*)
A class that handles 2D shear maps and allows to perform a set of operations on them

```
>>> from lenstools import ShearMap
```

```
>>> test = ShearMap.load("shear.fit",format=lenstools.defaults.load_fits_default_shear)
>>> test.side_angle
1.95
>>> test.data
#The actual map values
```

convergence()
Reconstructs the convergence from the E component of the shear

Returns new ConvergenceMap instance

eb_power_spectrum(*l_edges*)
Decomposes the shear map into its E and B modes components and returns the respective power spectral densities at the specified multipole moments

Parameters **l_edges** (*array*) – Multipole bin edges

Returns (l – array,P_EE,P_BB,P_EB – arrays) = (multipole moments, EE,BB power spectra and EB cross power)

Return type tuple.

```
>>> test_map = ShearMap.load("shear.fit",format=load_fits_default_shear)
>>> l_edges = np.arange(300.0,5000.0,200.0)
>>> l,EE,BB,EB = test_map.eb_power_spectrum(l_edges)
```

fourierEB()

Computes E and B modes of the shear map in Fourier space

Returns (E,B) map

Return type tuple.

fromEBmodes(*fourier_E*, *fourier_B*, *angle=<Quantity 3.14 deg>*)

This class method allows to build a shear map specifying its E and B mode components

Parameters

- **fourier_E** (*numpy 2D array, must be of type np.complex128 and must have a shape that is appropriate for a real fourier transform, i.e. (N,N/2 + 1); N should be a power of 2*) – E mode of the shear map in fourier space
- **fourier_B** (*numpy 2D array, must be of type np.complex128 and must have a shape that is appropriate for a real fourier transform, i.e. (N,N/2 + 1); N should be a power of 2*) – B mode of the shear map in fourier space
- **angle** (*float.*) – Side angle of the real space map in degrees

Returns the corresponding ShearMap instance

Raises AssertionErrors for inappropriate inputs

getValues(*x*, *y*)

Extract the map values at the requested (x,y) positions; this is implemented using the numpy fast indexing routines, so the formats of x and y must follow the numpy advanced indexing rules. Periodic boundary conditions are enforced

Parameters

- **x** (*numpy array or quantity*) – x coordinates at which to extract the map values (if unitless these are interpreted as radians)
- **y** (*numpy array or quantity*) – y coordinates at which to extract the map values (if unitless these are interpreted as radians)

Returns numpy array with the map values at the specified positions, with shape (N,shape x) where N is the number of components of the map field

Raises IndexError if the formats of x and y are not the proper ones

gradient(*x=None*, *y=None*)

Computes the gradient of the components of the spin1 field at each point

Parameters

- **x** (*array with units*) – optional, x positions at which to evaluate the gradient
- **y** (*array with units*) – optional, y positions at which to evaluate the gradient

Returns the gradient of the spin1 field in array form, of shape (4,:,:) where the four components are, respectively, 1x,1y,2x,2y; the units for the finite difference are pixels

info
Displays some of the information stored in the map (mainly resolution)

load (*filename*, *format=None*, ***kwargs*)
This class method allows to read the map from a data file, in various formats

Parameters

- **filename** (*str.*) – name of the file in which the map is saved
- **format** (*str. or callable*) – the format of the file in which the map is saved (can be a callable too); if None, it's detected automatically from the filename
- **kwargs** (*dict.*) – the keyword arguments are passed to the format (if callable)

Returns Spin1 instance with the loaded map

save (*filename*, *format=None*, *double_precision=False*)
Saves the map to an external file, of which the format can be specified (only fits implemented so far)

Parameters

- **filename** (*str.*) – name of the file on which to save the plane
- **format** (*str.*) – format of the file, only FITS implemented so far; if None, it's detected automatically from the filename
- **double_precision** (*bool.*) – if True saves the Plane in double precision

savefig (*filename*)
Saves the map visualization to an external file

Parameters **filename** (*str.*) – name of the file on which to save the map

setAngularUnits (*unit*)
Convert the angular units of the map to the desired unit

Parameters **unit** (*astropy units*) – astropy unit instance to which to perform the conversion

sticks (*fig=None*, *ax=None*, *pixel_step=10*, *multiplier=1.0*)
Draw the ellipticity map using the shear components

Parameters

- **ax** (*matplotlib ax object*) – ax on which to draw the ellipticity field
- **pixel_step** (*int.*) – One arrow will be drawn every pixel_step pixels to avoid arrow overplotting
- **multiplier** (*float.*) – Multiplies the stick length by a factor

Returns ax – the matplotlib ax object on which the stick field was drawn

```
>>> import matplotlib.pyplot as plt
>>> test = ShearMap.load("shear.fit",loader=load_fits_default_shear)
>>> fig,ax = plt.subplots()
>>> test.sticks(ax,pixel_step=50)
```

visualize (*fig=None*, *ax=None*, *component_labels=('$\\gamma_1$'*, *'$\\gamma_2$')*, *colorbar=False*, *cmap='jet'*, *cbar_label=None*, ***kwargs*)
Visualize the shear map; the kwargs are passed to imshow

visualizeComponents(*fig*, *ax*, *components='EE, BB, EB'*, *region=(200, 9000, -9000, 9000)*)
Plots the full 2D E and B mode power spectrum (useful to test statistical isotropicity)

Parameters

- **fig** (*matplotlibfigure object*) – figure on which to draw the ellipticity field
- **ax** (*matplotlib ax object or array of ax objects, can be None in which case new ax are created*) – ax on which to draw the ellipticity field
- **components** – string that contains the components to plot; the format is a sequence of {EE,BB,EB} separated by commas
- **region** (*tuple (lx_min, lx_max, ly_min, ly_max)*) – selects the multipole region to visualize

class lenstools.catalog.shear.**Catalog**(**args*, ***kwargs*)
Class handler of a galaxy catalogue, inherits all the functionality from the astropy.table.Table

pixelize(*map_size*, *npixel=256*, *field_quantity=None*, *origin=<Quantity [0., 0.] deg>*, *smooth=None*, *accumulate='average'*, *callback=None*, ***kwargs*)
Constructs a two dimensional square pixelized map version of one of the scalar properties in the catalog by assigning its objects on a grid

Parameters

- **map_size** (*quantity*) – spatial size of the map
- **npixel** (*int.*) – number of pixels on a side
- **field_quantity** (*str.*) – name of the catalog quantity to map; if None, 1 is assumed
- **origin** (*array with units*) – two dimensional coordinates of the origin of the map
- **smooth** (*quantity*) – if not None, the map is smoothed with a gaussian filter of scale smooth
- **accumulate** (*str.*) – if "sum" field galaxies that fall in the same pixel have their field_quantity summed, if "average" the sum is divided by the number of galaxies that fall in the pixel
- **callback** (*callable or None*) – user defined function that gets called on field_quantity
- **kwargs** (*dict.*) – the keyword arguments are passed to callback

Returns two dimensional scalar array with the pixelized field (pixels with no objects are treated as NaN)

Return type array

setSpatialInfo(*field_x='x'*, *field_y='y'*, *unit=Unit("deg")*)
Sets the spatial information in the catalog

Parameters

- **field_x** (*str.*) – name of the column that contains the x coordinates of the objects
- **field_y** (*str.*) – name of the column that contains the y coordinates of the objects
- **unit** (*astropy.unit*) – measure unit of the spatial coordinates

visualize(*map_size*, *npixel*, *field_quantity=None*, *origin=<Quantity [0., 0.] deg>*, *smooth=None*, *fig=None*, *ax=None*, *colorbar=False*, *cmap='jet'*, ***kwargs*)
Visualize a two dimensional square pixelized map version of one of the scalar properties in the catalog by assigning its objects on a grid (the pixelization is performed using the pixelize routine)

Parameters

- **map_size** (*quantity*) – spatial size of the map
- **npixel** (*int.*) – number of pixels on a side
- **field_quantity** (*str.*) – name of the catalog quantity to map; if None, 1 is assumed
- **origin** (*array with units*) – two dimensional coordinates of the origin of the map
- **smooth** (*quantity*) – if not None, the map is smoothed with a gaussian filter of scale smooth
- **kwargs** (*dict.*) – the additional keyword arguments are passed to pixelize

Returns two dimensional scalar array with the pixelized field (pixels with no objects are treated as NaN)

Return type array

class lenstools.catalog.shear.**ShearCatalog**(**args*, ***kwargs*)

Class handler of a galaxy shear catalog, inherits all the functionality from the Catalog class

shapeNoise(*seed=None*)

Generate a catalog with randomly drawn shape noise for each galaxy

Parameters **seed** (*int.*) – random seed for noise generation

Returns shape noise catalog

Return type ShearCatalog

toMap(*map_size*, *npixel*, *smooth*, ***kwargs*)

Convert a shear catalog into a shear map

Parameters

- **map_size** (*quantity*) – spatial size of the map
- **npixel** (*int.*) – number of pixels on a side
- **smooth** (*quantity*) – if not None, the map is smoothed with a gaussian filter of scale smooth
- **kwargs** (*dict.*) – additonal keyword arguments are passed to pixelize

Returns shear map

Return type *ShearMap*

9.1.3 Statistics

class lenstools.statistics.ensemble.**Ensemble**(*data=None*, *file_list=[]*, *metric='chi2'*, ***kwargs*)

A class that handles statistical operations on weak lensing maps, inherits from pandas DataFrame. The rows in the Ensemble correspond to the different ensemble realizations of the same descriptor

bootstrap(*callback*, *bootstrap_size=10*, *resample=10*, *seed=None*, *assemble=<built-in function array>*, *pool=None*, ***kwargs*)

Computes a custom statistic on the Ensemble using the bootstrap method

Parameters

- **callback** (*callable*) – statistic to compute on the ensemble; takes the resampled Ensemble data as an input

- **bootstrap_size** (*int.*) – size of the resampled ensembles used in the bootstraping; must be less than or equal to the number of realizations in the Ensemble
- **resample** (*int.*) – number of times the Ensemble is resampled
- **seed** (*int.*) – if not None, this is the random seed of the random resamples
- **assemble** (*callable*) – method that gets called on the resampled statistic list to make it into an Ensemble
- **pool** (*MPI pool object*) – MPI pool for multiprocessing (imported from emcee https://github.com/dfm/emcee)
- **kwargs** (*dict.*) – passed to the callback function

Returns the bootstraped statistic

Return type assemble return type

combine_columns(*combinations*)

Combine the hierarchical columns in the Ensemble, according to a dictionary which keys are the name of the combined features

Parameters **combinations** (*dict.*) – mapping of combined features onto the old ones

Returns Ensemble with columns combined

Return type Ensemble

classmethod combine_from_dict(*ensemble_dict*)

Builds an Ensemble combining the columns of smaller Ensembles; each key in the ensemble_dict dictionary becomes an element in the top level of the resulting Ensemble index

Parameters **ensemble_dict** (*dict.*) – dictionary that contains the Ensembles to combine; the values in the dictionary must be Ensembles

Return type Ensemble

compare(*rhs*, ***kwargs*)

Computes a comparison score between two Ensembles: computes a chi2-style difference between two different ensembles to assert how different they are

classmethod compute(*file_list*, *callback_loader=None*, *pool=None*, *index=None*, *assemble=<built-in function array>*, ***kwargs*)

Computes an ensemble, can spread the calculations on multiple processors using a MPI pool

Parameters

- **file_list** (*list.*) – list of files that will constitute the ensemble; the callback_loader is called on each of the files to produce the different realizations
- **callback_loader** (*function*) – This function gets executed on each of the files in the list and populates the ensemble. If None provided, it performs a numpy.load on the specified files. Must return a numpy array with the loaded data
- **pool** (*MPI pool object*) – MPI pool for multiprocessing (imported from emcee https://github.com/dfm/emcee)
- **index** (*pandas Index*) – index of the Ensemble
- **assemble** (*callable*) – called on the list of features (one feature per file) to assemble them in an array (defaults to np.array)
- **kwargs** (*dict.*) – Any additional keyword arguments to be passed to callback_loader

```
>>> from lenstools import Ensemble
>>> from lenstools.statistics import default_callback_loader
```

```
>>> map_list = ["conv1.fit","conv2.fit","conv3.fit"]
>>> l_edges = np.arange(200.0,50000.0,200.0)
```

```
>>> conv_ensemble = Ensemble.compute(map_list,callback_loader=default_callback_loader,pool=p
```

covariance (*bootstrap=False*, *bootstrap_size=10*, *resample=10*, *seed=None*, *pool=None*)

Computes the ensemble covariance matrix

Parameters

- **bootstrap** (*bool.*) – if True the covariance matrix is computed with a bootstrap estimate
- **bootstrap_size** (*int.*) – size of the resampled ensembles used in the bootstraping; must be less than or equal to the number of realizations in the Ensemble
- **seed** (*int.*) – if not None, this is the random seed of the random resamples
- **resample** (*int.*) – number of times the Ensemble is resampled
- **pool** (*MPI pool object*) – MPI pool for multiprocessing (imported from emcee https://github.com/dfm/emcee)

Returns ndarray with the covariance matrix, has shape (self.data[1],self.data[1])

group (*group_size*, *kind='sparse'*)

Sometimes it happens that different realizations in the ensemble need to be grouped together, for example when they belong to different subfields of the same observation field. With this function you can group different realizations together by taking the mean, and reduce the total number of realizations in the ensemble

Parameters

- **group_size** (*int*) – how many realizations to put in a group, must divide exactly the total number of realizations in the ensemble
- **kind** (*str.*) – specifies how to do the grouping; if set to "sparse" the groups are formed by taking one realizations every nobs/group_size (for example ([1,1001,...,9001],[2,1002,...,9002]) if nobs=10000 and group_size=10). If set to "contiguous" then the realizations are grouped as ([1,2,...,10],[11,12,...,20]). Otherwise you can set kind to your own sparse matrix scheme

Returns gropby object

imshow (*fig=None*, *ax=None*, ***kwargs*)

Visualize a two dimensional map of the Ensemble, with the index as the vertical axis and the columns as the horizontal axis

Parameters

- **fig** –
- **ax** –
- **kwargs** (*dict.*) –

classmethod meshgrid (*labels*, *sort=None*)

Construct on Ensemble whose column values are arranged in a regularly spaced mesh grid

Parameters

- **labels** (*dict.*) – dictionary whose keys are the Ensemble columns and whose values are the mesh grid axes values
- **sort** (*dict.*) – optional dictionary that tells how the meshgrid columns should be sorted

Return type Ensemble

principalComponents(*location=None*, *scale=None*)
Computes the principal components of the Ensemble

Parameters

- **location** (Series) – compute the principal components with respect to this location; must have the same columns as the Ensemble
- **scale** (Ensemble) – compute the principal components applying this scaling on the Ensemble columns; must have the same columns as the Ensemble

Returns pcaHandler instance

project(*vectors*, *names=None*)
Projects the rows of the Ensemble on the hyperplane defined by N linearly independent vectors

Parameters

- **vectors** (*tuple.*) – linearly independent vectors on which to project the rows of the Ensemble
- **names** (*list.*) – optional names of the projected components

Returns projected Ensemble; the new rows contain the components of the old rows along the vectors

Return type Ensemble

classmethod read(*filename*, *callback_loader=None*, ***kwargs*)
Reads a numpy file into an Ensemble

Parameters

- **filename** (*str.*) – name of the file to read
- **callback_loader** (*function*) – This function gets executed on each of the files in the list and populates the ensemble. If None provided, it unpickles the specified file. Must return an acceptable input for the Ensemble constructor
- **kwargs** (*dict.*) – Any additional keyword arguments to be passed to callback_loader

Returns Ensemble instance read from the file

classmethod readall(*filelist*, *callback_loader=None*, ***kwargs*)
Reads a list of files into an Ensemble

Parameters

- **filelist** (*list.*) – list of files to read
- **callback_loader** (*function*) – This function gets executed on each of the files in the list and populates the ensemble. If None provided, it performs a numpy.load on the specified file. Must return a numpy array with the loaded data
- **kwargs** (*dict.*) – Any additional keyword arguments to be passed to callback_loader

Returns Ensemble instance read from the file

save(*filename*, *format=None*, ***kwargs*)
Save ensemble data in an external file (in arbitrary format)

Parameters

- **filename** (*str.*) – file name of the external file
- **kwargs** (*dict.*) – the keyword arguments are passed to the saver (or to format if callable)

Format format in which to save the ensemble; if None the format is auto detected from the filename

selfChi2()

Computes the Ensemble distribution of chi squared values, defined with respect to the same Ensemble mean and covariance

Returns array with the self chi squared for each realization

shuffle(*seed=None*)

Changes the order of the realizations in the Ensemble

Parameters **seed** (*int.*) – random seed for the random shuffling

Returns shuffled Ensemble

suppress_indices(*by*, *suppress*, *columns*)

Combine multiple rows in the Ensemble into a single row, according to a specific criterion

Parameters

- **by** (*list.*) – list of columns that is used to group the Ensemble: the rows in each group are combined
- **suppress** (*list.*) – list of columns to suppress: these indices get suppressed when the rows are combined
- **columns** (*list.*) – list of columns to keep in the rows that are combined

Returns suppressed indices lookup table,combined Ensemble

Return type list.

class lenstools.statistics.database.**Database**(*name*)

insert(*df*, *table_name='data'*)

Parameters **df** (Ensemble) – records to insert in the database, in Ensemble (or pandas DataFrame) format

query(*sql*)

Parameters **sql** (*str.*) – sql query string

Returns Ensemble

classmethod **query_all**(*db_names*, *sql*)

Perform the same SQL query on a list of databases and combine the results

Parameters

- **db_names** (*list.*) – list of names of the databases to query
- **sql** (*str.*) – sql query string

Returns Ensemble

classmethod **read_table_all**(*db_names*, *table_name*)

Read the same SQL table from a list of databases and combine the results

Parameters

- **db_names** (*list.*) – list of names of the databases to query
- **table** (*str.*) – table to read

Returns Ensemble

class lenstools.statistics.database.**ScoreDatabase**(**args*, ***kwargs*)

insert(*df*, *table_name='data'*)

Parameters **df** (Ensemble) – records to insert in the database, in Ensemble (or pandas DataFrame) format

pull_features(*feature_list*, *table_name='scores'*, *score_type='likelihood'*)
Pull out the scores for a subset of features

Parameters

- **feature_list** (*list.*) – feature list to pull out from the database
- **score_type** (*str.*) – name of the column that contains the particular score you are considering

query(*sql*)

Parameters **sql** (*str.*) – sql query string

Returns Ensemble

query_all(*db_names*, *sql*)
Perform the same SQL query on a list of databases and combine the results

Parameters

- **db_names** (*list.*) – list of names of the databases to query
- **sql** (*str.*) – sql query string

Returns Ensemble

read_table_all(*db_names*, *table_name*)
Read the same SQL table from a list of databases and combine the results

Parameters

- **db_names** (*list.*) – list of names of the databases to query
- **table** (*str.*) – table to read

Returns Ensemble

lenstools.statistics.database.**chi2database**(**args*, ***kwargs*)
Populate an SQL database with the scores of different parameter sets with respect to the data; supports multiple features

Parameters

- **db_name** (*str.*) – name of the database to populate
- **parameters** (Ensemble) – parameter combinations to score
- **specs** (*dict.*) – dictionary that should contain the emulator,data, and covariance matrix of each feature to consider; each value in this dictionary must be a dictionary with keys 'emulator', 'data' and 'data covariance'

- **table_name** (*str.*) – table name to populate in the database
- **pool** (*MPIPool*) – MPIPool to spread the calculations over (pass None for automatic pool handling)
- **nchunks** (*int.*) – number of chunks to split the parameter score calculations in (one chunk per processor ideally)

9.1.4 Cosmological parameter estimation

class lenstools.statistics.constraints.**Analysis**(*data=None*, *file_list=[]*, *metric='chi2'*, ***kwargs*)

The base class of this module; the idea in weak lensing analysis is that one has a set of simulated data, that serves as training model, and then uses that set to fit the observations for the best model parameters. Inherits from Ensemble

add_models(*parameters*, *feature*)

Add a model to the training set of the current analysis

Parameters

- **parameters** (*array*) – parameter set of the new model
- **feature** (*array*) – measured feature of the new model

combine_features(*combinations*)

Combine features in the Analysis, according to a dictionary which keys are the name of the combined features

Parameters **combinations** (*dict.*) – mapping of combined features onto the old ones

find(*parameters*, *rtol=1e-05*)

Finds the location in the instance that has the specified combination of parameters

Parameters

- **parameters** (*array.*) – the parameters of the model to find
- **rtol** (*float.*) – tolerance of the search (must be less than 1)

Returns array of int. with the indices of the corresponding models

refeaturize(*transformation*, *method='apply_row'*, ***kwargs*)

Allows a general transformation on the feature set of the analysis by calling an arbitrary transformation function

Parameters

- **transformation** (*callable or dict.*) – callback function called on the feature_set; must take in a row of features and return a row of features. If a dictionary is passed, the keys must be the feature names
- **kwargs** (*dict.*) – the keyword arguments are passed to the transformation callable

Returns transformed Analysis

reparametrize(*transformation*, ***kwargs*)

Reparametrize the parameter set of the analysis by calling the formatter handle on the current parameter set (can be used to enlarge/shrink/relabel the parameter set)

Parameters

- **transformation** (*callable*) – transformation function called on the parameters, must take in a row of parameters and return another row of parameters

- **kwargs** (*dict.*) – the keyword arguments are passed to the transformation callable

Returns reparametrized Analysis

Fisher matrix calculations

class lenstools.statistics.constraints.**FisherAnalysis**(*data=None*, *file_list=[]*, *metric='chi2'*, ***kwargs*)

check()

Asserts that the parameters are varied one at a time, and that a parameter is not varied more than once

Raises AssertionError

chi2(*observed_feature*, *features_covariance*, *correct=None*)

Computes the chi2 between an observed feature and the fiducial feature, using the provided covariance

Parameters

- **observed_feature** (*array*) – observed feature to fit, its last dimension must have the same shape as self.feature_set[0]
- **features_covariance** (*2 dimensional array (or 1 dimensional if diagonal)*) – covariance matrix of the simulated features, must be provided for a correct fit!
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations

Returns chi2 of the comparison

Return type float.

classify(*observed_feature, features_covariance, correct=None, labels=[0, 1], confusion=False*)

Performs a Fisher classification of the observed feature, choosing the most probable label based on the value of the chi2

Parameters

- **observed_feature** (*array*) – observed feature to fit, the last dimenstion must have the same shape as self.feature_set[0]
- **features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, must be provided for a correct classification!
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations
- **labels** (*iterable*) – labels of the classification, must be the indices of the available classes (from 0 to feature_set.shape[0])
- **confusion** (*bool.*) – if True, an array with the label percentage occurrences is returned; if False an array of labels is returned

Returns array with the labels resulting from the classification

Return type int.

compute_derivatives()

Computes the feature derivatives with respect to the parameter sets using one step finite differences; the derivatives are computed with respect to the fiducial parameter set

Returns array of shape (p,N), where N is the feature dimension and p is the number of varied parameters

confidence_ellipse(*simulated_features_covariance, correct=None, observed_feature=None, observed_features_covariance=None, parameters=['Om', 'w'], p_value=0.684, **kwargs*)

Draws a confidence ellipse of a specified p-value in parameter space, corresponding to fit an observed feature for the cosmological parameters

Parameters

- **observed_feature** (*array*) – observed feature to fit, the last dimenstion must have the same shape as self.feature_set[0]
- **simulated_features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, must be provided for a correct fit!
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations
- **observed_features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, if different from the simulated one; if None the simulated feature covariance is used
- **parameters** (*list.*) – parameters to compute the condifence contour of
- **p_value** (*float.*) – p-value to calculate
- **kwargs** (*dict.*) – the keyword arguments are passed to the matplotlib Ellipse method

Returns matplotlib ellipse object

Return type Ellipse

ellipse(*center*, *covariance*, *p_value=0.684*, ***kwargs*)

Draws a confidence ellipse using matplotlib Ellipse patch

Parameters

- **center** (*tuple.*) – center of the ellipse
- **covariance** (*2D-array.*) – parameters covariance matrix
- **p_value** (*float.*) – p-value to calculate
- **kwargs** (*dict.*) – the keyword arguments are passed to the matplotlib Ellipse method

Returns matplotlib ellipse object

Return type Ellipse

fisher_matrix(*simulated_features_covariance*, *correct=None*, *observed_features_covariance=None*)

Computes the parameter Fisher matrix using the associated features, that in the end allows to compute the parameter confidence contours (around the fiducial value)

Parameters

- **simulated_features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, must be provided for a correct fit!
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations

- **observed_features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, if different from the simulated one; if None the simulated feature covariance is used

Returns 2 dimensional array with the Fisher matrix of the analysis

fit(*observed_feature*, *features_covariance*)

Maximizes the gaussian likelihood on which the Fisher matrix formalism is based, and returns the best fit for the parameters given the observed feature

Parameters

- **observed_feature** (*array*) – observed feature to fit, must have the same shape as self.feature_set[0]
- **features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, must be provided for a correct fit!

Returns array with the best fitted parameter values

parameter_covariance(*simulated_features_covariance*, *correct=None*, *observed_features_covariance=None*)

Computes the parameter covariance matrix using the associated features, that in the end allows to compute the parameter confidence contours (around the fiducial value)

Parameters

- **simulated_features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, must be provided for a correct fit!
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations
- **observed_features_covariance** (*2 dimensional array (or 1 dimensional if assumed diagonal)*) – covariance matrix of the simulated features, if different from the simulated one; if None the simulated feature covariance is used

Returns 2 dimensional array with the parameter covariance matrix of the analysis

set_fiducial(*n*)

Sets the fiducial model (with respect to which to compute the derivatives), default is 0 (i.e. self.parameter_set[0])

Parameters **n** (*int.*) – the parameter set you want to use as fiducial

variations

Checks the parameter variations with respect to the fiducial cosmology

Returns iterable with the positions of the variations

varied

Returns the indices of the parameters that are varied

Returns list with the indices of the varied parameters

where(*par=None*)

Finds the locations of the varied parameters in the parameter set

Returns dict. with the locations of the variations, for each parameter

Non linear parameter dependence

class lenstools.statistics.constraints.**Emulator**(**args*, ***kwargs*)

approximate_linear(*center*, *derivative_precision=0.1*)
Construct a FisherAnalysis by approximating the Emulator as a linear expansion along a chosen center

Parameters

- **center** (*Series*) – center point in parameter space
- **derivative_precision** (*float.*) – percentage step for the finite difference derivatives

Returns linearly approximated Fisher analysis

Return type *FisherAnalysis*

chi2(*parameters*, *observed_feature*, *features_covariance*, *correct=None*, *split_chunks=None*, *pool=None*)
Computes the chi2 part of the parameter likelihood with the usual sandwich product with the covariance matrix; the model features are computed with the interpolators

Parameters

- **parameters** (*(N,p) array where N is the number of points and p the number of parameters*) – new points in parameter space on which to compute the chi2 statistic
- **observed_feature** (*array*) – observed feature on which to condition the parameter likelihood
- **features_covariance** (*array*) – covariance matrix of the features, must be supplied
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations
- **split_chunks** (*int.*) – if set to an integer bigger than 0, splits the calculation of the chi2 into subsequent chunks, each that takes care of an equal number of points. Each chunk could be taken care of by a different processor

Returns array with the chi2 values, with the same shape of the parameters input

chi2Contributions(*parameters*, *observed_feature*, *features_covariance*, *correct=None*)
Computes the individual contributions of each feature bin to the chi2; the model features are computed with the interpolators. The full chi2 is the sum of the individual contributions

Parameters

- **parameters** (*(N,p) array where N is the number of points and p the number of parameters*) – new points in parameter space on which to compute the chi2 statistic
- **observed_feature** (*array*) – observed feature on which to condition the parameter likelihood
- **features_covariance** (*array*) – covariance matrix of the features, must be supplied
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations

Returns numpy 2D array with the contributions to the chi2 (off diagonal elements are the contributions of the cross correlation between bins)

likelihood(*chi2_value*, ***kwargs*)
Computes the likelihood value with the selected likelihood function, given the pre-computed chi2 value

Parameters

- **chi2_value** (*array*) – chi squared values
- **kwargs** – keyword arguments to be passed to your likelihood function

predict(*parameters*, *raw=False*)
Predicts the feature at a new point in parameter space using the bin interpolators, trained with the simulated features

Parameters

- **parameters** (*array*) – new points in parameter space on which to compute the chi2 statistic; it'a (N,p) array where N is the number of points and p the number of parameters, or array of size p if there is only one point
- **raw** (*bool.*) – if True returns raw numpy arrays

Returns predicted features

Return type array or Ensemble

sample_posterior(*observed_feature*, *sample='emcee'*, ***kwargs*)
Sample the parameter posterior distribution

Parameters

- **observed_feature** (*Series*) – observed feature to score
- **sample** (*str. or callable*) – posterior sampling method

Returns samples from the posterior distribution

Return type dict.

score(*parameters*, *observed_feature*, *method='chi2'*, ***kwargs*)
Compute the score for an observed feature for each combination of the proposed parameters

Parameters

- **parameters** (*DataFrame or array*) – parameter combinations to score
- **observed_feature** (*Series*) – observed feature to score
- **method** (*str. or callable*) – scoring method to use (defaults to chi2): if callable, must take in the current instance, the parameter array and the observed feature and return a score for each parameter combination
- **kwargs** (*dict.*) – keyword arguments passed to the callable method

Returns ensemble with the scores, for each feature

Return type Ensemble

set_likelihood(*function=None*)
Sets the likelihood function to a custom function input by the user: the default is the usual exp(-0.5*chi^2)

train(*use_parameters='all'*, *method='Rbf'*, ***kwargs*)
Builds the interpolators for each of the feature bins using a radial basis function approach

Parameters

- **use_parameters** (*list. or "all"*) – which parameters actually vary in the supplied parameter set (it doesn't make sense to interpolate over the constant ones)
- **method** (*str. or callable*) – interpolation method; can be 'Rbf' or callable. If callable, it must take two arguments, a square distance and a square length smoothing scale
- **kwargs** – keyword arguments to be passed to the interpolator constructor

Posterior samplers

lenstools.statistics.samplers.**emcee_sampler**(*emulator*, *observed_feature*, *features_covariance*, *correct=None*, *pslice=None*, *nwalkers=16*, *nburn=100*, *nchain=1000*, *pool=None*)

Parameter posterior sampling based on the MCMC algorithm implemented by the emcee package

Parameters

- **emulator** (Emulator) – feature emulator
- **observed_feature** (*array*) – observed feature to condition the parameter estimation
- **features_covariance** (*array*) – covariance matrix of the features
- **correct** (*int.*) – if not None, correct for the bias in the inverse covariance estimator assuming the covariance was estimated by 'correct' simulations
- **pslice** (*dict.*) – specify slices of the parameter space in which some parameters are keps as constants
- **nwalkers** (*int.*) – number of chains
- **nburn** (*int.*) – length of the burn-in chain
- **nchain** (*int.*) – length of the MCMC chain
- **pool** (*MPIPool*) – MPI Pool for parallelization of computations

Returns ensemble of samples from the posterior probability distribution

Return type Ensemble

Confidence contour plotting

class lenstools.statistics.contours.**ContourPlot**(*fig=None*, *ax=None*)

A class handler for contour plots

close()

Closes the figure

confidenceArea()

Computes the area enclosed by the parameter confidence contours

Returns area enclosed by the confidence contours

Return type dict.

expectationValue(*function*, ***kwargs*)

Computes the expectation value of a function of the parameters over the current parameter likelihood

classmethod from_scores(*score_ensemble*, *parameters*, *feature_names=None*, *plot_labels=None*, *fig=None*, *ax=None*, *figsize=(8, 8)*)

Build a ContourPlot instance out of an ensemble of scores

Parameters

- **score_ensemble** (Ensemble) – ensemble of the scores
- **parameters** (*list.*) – columns names that contain the parameters
- **feature_names** (*list.*) – name of the features to generate the contour plot for
- **plot_labels** (*list.*) – plot labels for the parameters
- **figsize** (*tuple.*) – size of the plot

Returns list of ContourPlot

getLikelihood(*likelihood_filename*, *parameter_axes={'w': 1, 'Omega_m': 0, 'sigma8': 2}*, *parameter_labels={'w': 'w', 'Omega_m': '$\\Omega_m$', 'sigma8': '$\\sigma_8$'}*)
Load the likelihood function from a numpy file

getLikelihoodValues(*levels*, *precision=0.001*)
Find the likelihood values that correspond to the selected p_values

getMaximum(*which='full'*)
Find the point in parameter space on which the likelihood is maximum

getUnitsFromOptions(*options*)
Parse options file to get physical units of axes

labels(*contour_label=None*, *fontsize=22*, ***kwargs*)
Put the labels on the plot

marginal(*parameter_name='w'*, *levels=None*)
Marginalize the likelihood over all parameters but one

marginalize(*parameter_name='w'*)
Marginalize the likelihood over the indicated parameters

plotContours(*colors=['red', 'green', 'blue'], display_percentages=True, display_maximum=True, color_maximum='green', fill=False, **kwargs*)
Display the confidence likelihood contours

plotEllipse(*colors=['red'], levels=[0.683], **kwargs*)
Plot the Gaussian approximation to the confidence contours

Parameters

- **colors** (*list.*) – colors of the confidence ellipses
- **levels** (*list.*) – p_values of the confidence ellipses
- **kwargs** (*dict.*) – additional keyword arguments are passed to the Ellipse patch method

plotMarginal(*parameter, levels=[0.684], colors=['red', 'blue', 'green'], alpha=0.5, fill=False*)
Plot the likelihood function marginalized over all parameters except one

point(*coordinate_x*, *coordinate_y*, *color='green'*, *marker='o'*)
Draws a point in parameter space at the specified physical coordinates

savefig(*figname*)
Save the plot to file

setUnits(*parameter*, *parameter_min*, *parameter_max*, *parameter_unit*)
Set manually the physical units for each of the likelihood axes

show(*cmap=None*, *interpolation='nearest'*)
Show the 2D marginalized likelihood

slice (*parameter_name='w'*, *parameter_value=-1.0*)
Slice the likelihood cube by fixing one of the parameters

value (**coordinates*)
Compute the (un-normalized) likelihood value at the specified point in parameter space

variance (*function*, ***kwargs*)
Computes the variance of a function of the parameters over the current parameter likelihood

9.1.5 Noise

class lenstools.image.noise.**GaussianNoiseGenerator** (*shape*, *side_angle*, *label*)
A class that handles generation of Gaussian simulated noise maps

classmethod forMap (*conv_map*)
This class method generates a Gaussian noise generator intended to be used on a convergence map: i.e. the outputs of its methods can be added to the convergence map in question to simulate the presence of noise

Parameters **conv_map** (*ConvergenceMap instance*) – The blueprint of the convergence map you want to generate the noise for

Raises AssertionError if conv_map is not a ConvergenceMap instance

fromConvPower (*power_func*, *seed=0*, ***kwargs*)
This method uses a supplied power spectrum to generate correlated noise maps in real space via FFTs

Parameters

- **power_func** (*function with the above specifications, or numpy array (l,Pl) of shape (2,n)*) – function that given a numpy array of l's returns a numpy array with the according Pl's (this is the input power spectrum); alternatively you can pass an array (l,Pl) and the power spectrum will be calculated with scipy's interpolation routines
- **seed** (*int.*) – seed of the random generator
- **kwargs** – keyword arguments to be passed to power_func, or to the interpolate.interp1d routine

Returns ConvergenceMap instance of the same exact shape as the one used as blueprint

getShapeNoise (*z=1.0*, *ngal=<Quantity 15.0 1 / arcmin2>*, *seed=0*)
This method generates a white, gaussian shape noise map for the given redshift of the map

Parameters

- **z** (*float.*) – single redshift of the backround sources on the map
- **ngal** (*float.*) – assumed angular number density of galaxies (must have units of angle^-2)
- **seed** (*int.*) – seed of the random generator

Returns ConvergenceMap instance of the same exact shape as the one used as blueprint

9.1.6 Existing Simulation suites

class lenstools.simulations.**IGS1** (*H0=72.0*, *Om0=0.26*, *w0=-1.0*, *sigma8=0.798*, *ns=0.96*, *root_path=None*, *name=None*)
Class handler of the IGS1 simulations set, inherits the cosmological parameters from the astropy.cosmology.FlatwCDM class; the default parameter values are the fiducial ones

getNames (*realizations*, *z=1.0*, *kind='convergence'*, *big_fiducial_set=False*)
Get the full name of the IGS1 maps, once a redshift, realization identificator and kind are specified

Parameters

- **z** (*float.*) – redshift plane of the maps, must be one of [1.0,1.5,2.0]
- **realizations** (*list. or int.*) – list of realizations to get the names of, the elements must be in [1,1000]
- **kind** (*str.*) – decide if retrieve convergence or shear maps, must be one of [convergence,shear1,shear2]
- **big_fiducial_set** (*bool.*) – set to True if you want to get the names of the bigger fiducial simulation based on 45 N-body simulations

load (*realization*, ***kwargs*)
Reads in a specific realization of the convergence field (in FITS format) and returns a ConvergenceMap instance with the loaded map

Parameters

- **realization** (*int.*) – the specific realization to read
- **kwargs** – the keyword arguments are passed to the getNames method

Returns ConvergenceMap instance with the loaded map

squeeze (*with_ns=False*)
Returns the cosmological parameters of the model in numpy array form

Parameters **with_ns** (*bool.*) – if True returns also ns as the last parameter

Returns numpy array (Om0,w0,si8,ns–optionally)

class lenstools.simulations.**CFHTemu1** (*H0=72.0*, *Om0=0.26*, *w0=-1.0*, *sigma8=0.798*, *ns=0.96*, *root_path=None*, *name=None*)
Class handler of the weak lensing CFHTemu1 simulations set, inherits from IGS1; this simulation suite contains 91 different cosmological models based on 1 N-body simulation each. Each model has 1000 realizations for each of the 13 CFHT subfields

classmethod getModels (*root_path='/default'*)
This class method uses a dictionary file to read in all the cosmological model parameters and instantiate the corresponding CFHTemu1 objects for each one of them

Parameters **root_path** (*str.*) – path of your CFHT emu1 simulations copy

Returns list of CFHTemu1 instances

getNames (*realizations*, *subfield=1*, *smoothing=0.5*)
Get the full name of the CFHT emu1 maps, once a subfield and smoothing scale are specified

Parameters

- **subfield** (*int.*) – the specific CFHT subfield you want to retrieve, must be between 1 and 13
- **smoothing** (*float.*) – smoothing scale of the maps you wish to retrieve, in arcmin
- **realizations** (*list. or int.*) – list of realizations to get the names of, the elements must be in [1,1000]

load (*realization*, ***kwargs*)
Reads in a specific realization of the convergence field (in FITS format) and returns a ConvergenceMap instance with the loaded map

Parameters

- **realization** (*int.*) – the specific realization to read
- **kwargs** – the keyword arguments are passed to the getNames method

Returns ConvergenceMap instance with the loaded map

squeeze(*with_ns=False*)
Returns the cosmological parameters of the model in numpy array form

Parameters **with_ns** (*bool.*) – if True returns also ns as the last parameter

Returns numpy array (Om0,w0,si8,ns–optionally)

class lenstools.simulations.**CFHTcov**(*H0=72.0*, *Om0=0.26*, *w0=-1.0*, *sigma8=0.798*, *ns=0.96*, *root_path=None*, *name=None*)
Class handler of the weak lensing CFHTcov simulations set, inherits from CFHTemu1; this simulation suite contains 1000 realizations for each of the 13 CFHT subfields, based on 50 independent N-body simulations of a fiducial LambdaCDM universe. Useful to measure accurately descriptor covariance matrices

classmethod getModels(*root_path='/default'*)
On call, this class method returns a CFHTcov instance initialized with the cosmological parameters of the only available model in the suite

Parameters **root_path** (*str.*) – path of your CFHTcov simulations copy

Returns CFHTcov instance initialized with the fiducial cosmological parameters

getNames(*realizations*, *subfield=1*, *smoothing=0.5*)
Get the full name of the CFHTcov maps, once a subfield and smoothing scale are specified

Parameters

- **subfield** (*int.*) – the specific CFHT subfield you want to retrieve, must be between 1 and 13
- **smoothing** (*float.*) – smoothing scale of the maps you wish to retrieve, in arcmin
- **realizations** (*list. or int.*) – list of realizations to get the names of, the elements must be in [1,1000]

load(*realization*, ***kwargs*)
Reads in a specific realization of the convergence field (in FITS format) and returns a ConvergenceMap instance with the loaded map

Parameters

- **realization** (*int.*) – the specific realization to read
- **kwargs** – the keyword arguments are passed to the getNames method

Returns ConvergenceMap instance with the loaded map

squeeze(*with_ns=False*)
Returns the cosmological parameters of the model in numpy array form

Parameters **with_ns** (*bool.*) – if True returns also ns as the last parameter

Returns numpy array (Om0,w0,si8,ns–optionally)

9.1.7 Simulation design

class lenstools.simulations.**Design**(*data=None*, *file_list=[]*, *metric='chi2'*, ***kwargs*)
A class that proves useful in designing simulation sets: the main functionality provided is the uniform sampling

of an arbirtarily high dimensional parameter space. The points in parameter space are chosen to be as spread as possible by minimizing a cost function, but enforcing a latin hypercube structure, in which each parameter value appears once

cost (*p*, *Lambda*)
Computes the cost function of the current configuration given the metric parameters (p,Lambda)

Parameters

- **Lambda** (*float.*) – metric parameter of the cost function; if set to 1.0 the cost function corresponds is the Coulomb potential energy
- **p** (*float.*) – metric parameter of the cost function; if set to 2.0 the distances between points are the Euclidean ones

Returns the value of the cost function

diagonalCost (*Lambda*)
Computes the cost function of a diagonal configuration with a specified number of points and a metric parameter lambda; the cost function is calculated on the scaled parameter values, which are always between 0 and 1

Parameters **Lambda** (*float.*) – metric parameter of the cost function; if set to 1.0 the cost function corresponds is the Coulomb potential energy

Returns the value of the cost function for a diagonal configuration

classmethod from_specs (*npoints*, *parameters*)

Parameters

- **npoints** (*int.*) – number of points in the design
- **parameters** (*list.*) – list of tuples (name,label,min,max)

sample (*p=2.0*, *Lambda=1.0*, *seed=0*, *maxIterations=10000*)
Evenly samples the parameter space by minimizing the cost function computed with the metric parameters (p,Lambda); this operation works inplace

Parameters

- **Lambda** (*float.*) – metric parameter of the cost function; if set to 1.0 the cost function corresponds is the Coulomb potential energy
- **p** (*float.*) – metric parameter of the cost function; if set to 2.0 the distances between points are the Euclidean ones
- **seed** (*int.*) – random seed with which the sampler random generator will be initialized
- **maxIterations** (*int.*) – maximum number of iterations that the sampler can perform before stopping

Returns the relative change of the cost function the last time it varied during the sampling

classmethod sample_ellipse (*npoints*, *parameters*, *center*, *minor*, *major*, *angle*, *radial_power=0.5*, ***kwargs*)
Sample points in a two dimensional parameter space whose boundary has the shape of an ellipse

Parameters

- **npoints** (*int.*) – number of points in the design
- **parameters** (*list.*) – name of the parameters, the list must have 2 elements
- **center** (*tuple.*) – center of the ellipse
- **minor** (*float.*) – length of the semi-minor axis

- **major** (*float.*) – length of the semi-major axis
- **angle** (*float.*) – rotation angle of the major axis with respect to the horizontal (degrees counterclockwise)
- **radial_power** (*float.*) – this parameter controls the density of points around the center (higher for higher density); 0.5 corresponds to uniform sampling
- **kwargs** – the keyword arguments are passed to the sample() method

Returns

Return type Design

savefig(*filename*)
Save the visualization to an external file

Parameters **filename** (*str.*) – name of the file on which to save the plot

set_title(*title*)
Give a title to the visualized design

Parameters **title** (*str.*) – title of the figure

visualize(*fig=None*, *ax=None*, *parameters=None*, ***kwargs*)
Visualize the design configuration using matplotlib

Parameters **parameters** (*list.*) – the parameters to visualize, you can specify two or three of them, by their names. If None, all parameters are visualized

write(*filename=None*, *max_rows=None*, *format='ascii.latex'*, *column_format='{0:.3f}'*, ***kwargs*)
Outputs the points that make up the design in a nicely formatted table

Parameters

- **filename** (*str. or file descriptor*) – name of the file to which the table will be saved; if None the contents will be printed
- **max_rows** (*int.*) – maximum number of rows in the table, if smaller than the number of points the different chunks are hstacked (useful if there are too many rows for display)
- **format** (*str.*) – passed to the Table.write astropy method
- **column_format** (*str.*) – format specifier for the numerical values in the Table
- **kwargs** (*dict.*) – the keyword arguments are passed to astropy.Table.write method

Returns the Table instance with the design parameters

9.1.8 Nicaea bindings

class lenstools.simulations.**NicaeaSettings**
Class handler of the code settings (non linear modeling, tomography, transfer function, etc...)

available(*knob*)
Given a settings, lists all the possible values

classmethod **default**()
Generate default settings

Returns NicaeaSettings defaults instance

knobs
Lists available settings to tune

class lenstools.simulations.**Nicaea**(*H0=72.0*, *Om0=0.26*, *Ode0=0.74*, *Ob0=0.046*, *w0=-1.0*, *wa=0.0*, *sigma8=0.8*, *ns=0.96*, *name=None*)

Main class handler for the python bindings of the NICAEA cosmological code, written by M. Kilbinger & collaborators

convergencePowerSpectrum(*ell*, *z=2.0*, *distribution=None*, *distribution_parameters=None*, *settings=None*, ***kwargs*)

Computes the convergence power spectrum for the given cosmological parameters and redshift distribution using NICAEA

Parameters

- **ell** (*array.*) – multipole moments at which to compute the power spectrum
- **z** (*float., array or None*) – redshift bins for the sources; if a single float is passed, single redshift is assumed
- **distribution** (*None,callable or list*) – redshift distribution of the sources (normalization not necessary); if None a single redshift is expected; if callable, z must be an array, and a single redshift bin is considered, with the galaxy distribution specified by the call of distribution(z); if a list is passed, each element must be a NICAEA type
- **distribution_parameters** (*str. or list.*) – relevant only when distribution is a list or callable. When distribution is callable, distribution_parameters has to be one between "one" and "all" to decide if one or multiple redshift bins have to be considered. If it is a list, each element in it should be the tuple of parameters expected by the correspondent NICAEA distribution type
- **settings** (*NicaeaSettings instance*) – NICAEA code settings
- **kwargs** (*dict.*) – the keyword arguments are passed to the distribution, if callable

Returns (NlxNz array) computed power spectrum at the selected multipoles (when computing the cross components these are returned in row major C ordering)

classmethod fromCosmology(*cosmo*)

Builds a Nicaea instance from one of astropy.cosmology objects, from which it inherits all the cosmological parameter values

Parameters **cosmo** (*astropy FLRW*) – one of astropy cosmology instances

Returns Nicaea instance with the cosmological parameters inherited from cosmo

shearTwoPoint(*theta*, *z=2.0*, *distribution=None*, *distribution_parameters=None*, *settings=None*, *kind='+'*, ***kwargs*)

Computes the shear two point function for the given cosmological parameters and redshift distribution using NICAEA

Parameters

- **theta** (*array. with units*) – angles at which to compute the two point function
- **z** (*float., array or None*) – redshift bins for the sources; if a single float is passed, single redshift is assumed
- **distribution** (*None,callable or list*) – redshift distribution of the sources (normalization not necessary); if None a single redshift is expected; if callable, z must be an array, and a single redshift bin is considered, with the galaxy distribution specified by the call of distribution(z); if a list is passed, each element must be a NICAEA type
- **distribution_parameters** (*str. or list.*) – relevant only when distribution is a list or callable. When distribution is callable, distribution_parameters has to be

one between "one" and "all" to decide if one or multiple redshift bins have to be considered. If it is a list, each element in it should be the tuple of parameters expected by the correspondent NICAEA distribution type

- **settings** (*NicaeaSettings instance*) – NICAEA code settings
- **kind** (*str.*) – must be "+" or "-"
- **kwargs** (*dict.*) – the keyword arguments are passed to the distribution, if callable

Returns (NtxNz array) computed two point function at the selected angles (when computing the cross components these are returned in row major C ordering)

9.1.9 N-body simulation snapshot handling

class lenstools.simulations.**Gadget2Snapshot**(*fp=None*, *pool=None*, *length_unit=<Quantity 1.0 kpc>*, *mass_unit=<Quantity 10000000000.0 solMass>*, *velocity_unit=<Quantity 1.0 km / s>*, *header_kwargs={}*)

A class that handles Gadget2 snapshots, mainly I/O from the binary format and spatial information statistics.Inherits from the abstract NbodySnapshot

close()

Closes the snapshot file

cutPlaneAdaptive(*normal=2*, *center=<Quantity 7.0 Mpc>*, *left_corner=None*, *plane_resolution=<Quantity 0.1 Mpc>*, *neighbors=64*, *neighborDistances=None*, *kind='density'*, *projectAll=False*)

Cuts a density (or gravitational potential) plane out of the snapshot by computing the particle number density using an adaptive smoothing scheme; the plane coordinates are cartesian comoving

Parameters

- **normal** (*int. (0,1,2)*) – direction of the normal to the plane (0 is x, 1 is y and 2 is z)
- **center** (*float. with units*) – location of the plane along the normal direction
- **plane_resolution** (*float. with units (or int.)*) – plane resolution (perpendicular to the normal)
- **left_corner** (*tuple of quantities or None*) – specify the position of the lower left corner of the box; if None, the minimum of the (x,y,z) of the contained particles is assumed
- **neighbors** (*int.*) – number of nearest neighbors to use in the adaptive smoothing procedure
- **neighborDistances** (*array with units*) – precomputed distances of each particle to its N-th nearest neighbor; if None these are computed
- **kind** (*str. ("density" or "potential")*) – decide if computing a density or gravitational potential plane (this is computed solving the poisson equation)
- **projectAll** (*bool.*) – if True, all the snapshot is projected on a single slab perpendicular to the normal, ignoring the position of the center

Returns tuple(numpy 2D array with the computed particle number density (or lensing potential),bin resolution along the axes,number of particles on the plane)

cutPlaneAngular(*normal=2*, *thickness=<Quantity 0.5 Mpc>*, *center=<Quantity 7.0 Mpc>*, *left_corner=None*, *plane_lower_corner=<Quantity [0., 0.] deg>*, *plane_size=<Quantity 0.15 deg>*, *plane_resolution=<Quantity 1.0 arcmin>*, *thickness_resolution=<Quantity 0.1 Mpc>*, *smooth=None*, *tomography=False*, *kind='density'*, *space='real'*)

Same as cutPlaneGaussianGrid(), except that this method will return a lens plane as seen from an observer at z=0; the spatial transverse units are converted in angular units as seen from the observer

Parameters

- **normal** (*int. (0,1,2)*) – direction of the normal to the plane (0 is x, 1 is y and 2 is z)
- **thickness** (*float. with units*) – thickness of the plane
- **center** (*float. with units*) – location of the plane along the normal direction; it is assumed that the center of the plane is seen from an observer with a redshift of self.header["redshift"]
- **left_corner** (*tuple of quantities or None*) – specify the position of the lower left corner of the box; if None, the minimum of the (x,y,z) of the contained particles is assumed
- **plane_lower_corner** (*float with units.*) – lower left corner of the plane, as seen from the observer (0,0) corresponds to the lower left corner of the snapshot
- **plane_size** (*float with units*) – angular size of the lens plane (angles start from 0 in the lower left corner)
- **plane_resolution** (*float. with units (or int.)*) – plane angular resolution (perpendicular to the normal)
- **thickness_resolution** (*float. with units (or int.)*) – plane resolution (along the normal)
- **smooth** (*int. or None*) – if not None, performs a smoothing of the angular density (or potential) with a gaussian kernel of scale "smooth x the pixel resolution"
- **tomography** (*bool.*) – if True returns the lens plane angular density for each slab, otherwise a projected density (or lensing potential) is computed
- **kind** (*str. ("density" or "potential")*) – decide if computing an angular density or lensing potential plane (this is computed solving the poisson equation)
- **space** (*str.*) – if "real" return the lens plane in real space, if "fourier" the Fourier transform is not inverted

Returns tuple(numpy 2D or 3D array with the (unsmoothed) particle angular number density,bin angular resolution, total number of particles on the plane); the constant spatial part of the density field is subtracted (we keep the fluctuation only)

cutPlaneGaussianGrid(*normal=2*, *thickness=<Quantity 0.5 Mpc>*, *center=<Quantity 7.0 Mpc>*, *plane_resolution=<Quantity 0.1 Mpc>*, *left_corner=None*, *thickness_resolution=<Quantity 0.1 Mpc>*, *smooth=None*, *kind='density'*, ***kwargs*)

Cuts a density (or gravitational potential) plane out of the snapshot by computing the particle number density on a slab and performing Gaussian smoothing; the plane coordinates are cartesian comoving

Parameters

- **normal** (*int. (0,1,2)*) – direction of the normal to the plane (0 is x, 1 is y and 2 is z)

- **thickness** (*float. with units*) – thickness of the plane
- **center** (*float. with units*) – location of the plane along the normal direction
- **plane_resolution** (*float. with units (or int.)*) – plane resolution (perpendicular to the normal)
- **left_corner** (*tuple of quantities or None*) – specify the position of the lower left corner of the box; if None, the minimum of the (x,y,z) of the contained particles is assumed
- **thickness_resolution** (*float. with units (or int.)*) – plane resolution (along the normal)
- **smooth** (*int. or None*) – if not None, performs a smoothing of the density (or potential) with a gaussian kernel of scale "smooth x the pixel resolution"
- **kind** (*str. ("density" or "potential")*) – decide if computing a density or gravitational potential plane (this is computed solving the poisson equation)
- **kwargs** (*dict.*) – accepted keyword are: 'density_placeholder', a pre-allocated numpy array, with a RMA window opened on it; this facilitates the communication with different processors by using a single RMA window during the execution. 'l_squared' a pre-computed meshgrid of squared multipoles used for smoothing

Returns tuple(numpy 3D array with the (unsmoothed) particle number density,bin resolution along the axes, number of particles on the plane)

getID (*first=None*, *last=None*, *save=True*)

Reads in the particles IDs, 4 byte ints, (read in of a subset is allowed): when first and last are specified, the numpy array convention is followed (i.e. getID(first=a,last=b)=getID()[a:b])

Parameters

- **first** (*int. or None*) – first particle in the file to be read, if None 0 is assumed
- **last** (*int. or None*) – last particle in the file to be read, if None the total number of particles is assumed
- **save** (*bool.*) – if True saves the particles IDs as attribute

Returns numpy array with the particle IDs

getPositions (*first=None*, *last=None*, *save=True*)

Reads in the particles positions (read in of a subset is allowed): when first and last are specified, the numpy array convention is followed (i.e. getPositions(first=a,last=b)=getPositions()[a:b])

Parameters

- **first** (*int. or None*) – first particle in the file to be read, if None 0 is assumed
- **last** (*int. or None*) – last particle in the file to be read, if None the total number of particles is assumed
- **save** (*bool.*) – if True saves the particles positions as attribute

Returns numpy array with the particle positions

getVelocities (*first=None*, *last=None*, *save=True*)

Reads in the particles velocities (read in of a subset is allowed): when first and last are specified, the numpy array convention is followed (i.e. getVelocities(first=a,last=b)=getVelocities()[a:b])

Parameters

- **first** (*int. or None*) – first particle in the file to be read, if None 0 is assumed

- **last** (*int. or None*) – last particle in the file to be read, if None the total number of particles is assumed
- **save** (*bool.*) – if True saves the particles velocities as attrubute

Returns numpy array with the particle velocities

gridID()
Compute an ID for the particles in incresing order according to their position on a Nside x Nside x Nside grid; the id is computed as x + y*Nside + z*Nside**2

Returns the gridded IDs

Return type array of float

header
Displays the snapshot header information

Returns the snapshot header information in dictionary form

Return type dict.

lensMaxSize()
Computes the maximum observed size of a lens plane cut out of the current snapshot

massDensity(*resolution=<Quantity 0.5 Mpc>*, *smooth=None*, *left_corner=None*, *save=False*)
Uses a C backend gridding function to compute the matter mass density fluctutation for the current snapshot: the density is evaluated using a nearest neighbor search

Parameters

- **resolution** (*float with units or int.*) – resolution below which particles are grouped together; if an int is passed, this is the size of the grid
- **smooth** (*int. or None*) – if not None, performs a smoothing of the density (or potential) with a gaussian kernel of scale "smooth x the pixel resolution"
- **left_corner** (*tuple of quantities or None*) – specify the position of the lower left corner of the box; if None, the minimum of the (x,y,z) of the contained particles is assumed
- **save** (*bool.*) – if True saves the density histogram and resolution as instance attributes

Returns tuple(numpy 3D array with the (unsmoothed) matter density fluctuation on a grid,bin resolution along the axes)

neighborDistances(*neighbors=64*)
Find the N-th nearest neighbors to each particle

Parameters **neighbors** (*int.*) – neighbor order

Returns array with units

open(*filename*, *pool=None*, *header_kwargs={}*, ***kwargs*)
Opens a snapshot at filename

Parameters

- **filename** (*str. or file.*) – file name of the snapshot
- **pool** (*MPIWhirlPool instance*) – use to distribute the calculations on different processors; if not None, each processor takes care of one of the snapshot parts, appending as ".n" to the filename
- **header_kwargs** (*dict.*) – keyword arguments to pass to the getHeader method

- **kwargs** (*dict.*) – the keyword arguments are passed to buildFilename

pos2R(*filename*, *variable_name='pos'*)
Saves the positions of the particles in a R readable format, for facilitating visualization with RGL

Parameters

- **filename** (*str.*) – name of the file on which to save the particles positions
- **variable_name** (*str.*) – name of the variable that contains the (x,y,z) positions in the R environment

powerSpectrum(*k_edges*, *resolution=None*, *return_num_modes=False*)
Computes the power spectrum of the relative density fluctuations in the snapshot at the wavenumbers specified by k_edges; a discrete particle number density is computed before hand to prepare the FFT grid

Parameters

- **k_edges** (*array.*) – wavenumbers at which to compute the density power spectrum (must have units)
- **resolution** (*float with units, int. or None*) – optional, fix the grid resolution to some value; to be passed to the massDensity method. If none this is computed automatically from the k_edges
- **return_num_modes** (*bool.*) – if True returns the mode counting for each k bin as the last element in the return tuple

Returns tuple(k_values(bin centers),power spectrum at the specified k_values)

reorder()
Sort particles attributes according to their ID

savefig(*filename*)
Save the snapshot visualization to an external file

Parameters **filename** (*str.*) – file name to which the figure will be saved

setHeaderInfo(*Om0=0.26, Ode0=0.74, w0=-1.0, wa=0.0, h=0.72, redshift=100.0, box_size=<Quantity 20.833333333333336 Mpc>, flag_cooling=0, flag_sfr=0, flag_feedback=0, flag_stellarage=0, flag_metals=0, flag_entropy_instead_u=0, masses=<Quantity [0.00000000e+00, 1.03000000e+10, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00] solMass>, num_particles_file_of_type=None, npartTotalHighWord=array([0, 0, 0, 0, 0, 0], dtype=uint32)*)
Sets the header info in the snapshot to write

setPositions(*positions*)
Sets the positions in the current snapshot (with the intent of writing them to a properly formatted snapshot file)

Parameters **positions** (*(N,3) array with units*) – positions of the particles, must have units

setVelocities(*velocities*)
Sets the velocities in the current snapshot (with the intent of writing them to a properly formatted snapshot file)

Parameters **velocities** (*(N,3) array with units*) – velocities of the particles, must have units

visualize(*fig=None*, *ax=None*, *scale=False*, *first=None*, *last=None*, ***kwargs*)
Visualize the particles in the snapshot using the matplotlib 3D plotting engine, the kwargs are passed to the matplotlib scatter method

Parameters **scale** (*bool.*) – if True, multiply all the (comoving) positions by the scale factor

write (*filename*, *files=1*)
Writes particles information (positions, velocities, etc...) to a properly formatter Gadget snapshot

Parameters

- **filename** (*str.*) – name of the file to which to write the snapshot
- **files** (*int.*) – number of files on which to split the writing of the snapshot (useful if the number of particles is large); if > 1 the extension ".n" is appended to the filename

writeParameterFile (*filename*, *settings=<lenstools.simulations.gadget2.Gadget2Settings object>*)
Writes a Gadget2 parameter file to evolve the current snapshot using Gadget2

Parameters

- **filename** (*str.*) – name of the file to which to write the parameters
- **settings** (Gadget2Settings) – tunable settings of Gadget2 (see Gadget2 manual)

class lenstools.simulations.gadget2.**Gadget2SnapshotDE** (*fp=None*, *pool=None*, *length_unit=<Quantity 1.0 kpc>*, *mass_unit=<Quantity 10000000000.0 solMass>*, *velocity_unit=<Quantity 1.0 km / s>*, *header_kwargs={}*)
A class that handles Gadget2 snapshots, mainly I/O from the binary format and spatial information statistics.Inherits from Gadget2Snapshot; assumes that the header includes Dark Energy information

class lenstools.simulations.gadget2.**Gadget2SnapshotPipe** (**args*, ***kwargs*)
Read in the particle positions when calling the constructor, without calling fseek

class lenstools.simulations.amiga.**AmigaHalos** (*fp=None*, *pool=None*, *length_unit=<Quantity 1.0 kpc>*, *mass_unit=<Quantity 10000000000.0 solMass>*, *velocity_unit=<Quantity 1.0 km / s>*, *header_kwargs={}*)
A class that handles the Amiga Halo Finder (AHF) halo output files. Inherits from the abstract NbodySnapshot

Warning Not tested yet!

9.1.10 Ray tracing simulations

class lenstools.simulations.**Plane** (*data*, *angle*, *redshift=2.0*, *cosmology=None*, *comoving_distance=None*, *unit=Unit("rad2")*, *num_particles=None*, *masked=False*, *filename=None*)

classmethod **load** (*filename*, *format=None*, *init_cosmology=True*)
Loads the Plane from an external file, of which the format can be specified (only fits implemented so far)

Parameters

- **filename** (*str.*) – name of the file from which to load the plane
- **format** (*str.*) – format of the file, only FITS implemented so far; if None, it's detected automatically from the filename
- **init_cosmology** (*bool.*) – if True, instantiates the cosmology attribute of the PotentialPlane

Returns PotentialPlane instance that wraps the data contained in the file

randomRoll(*seed=None*, *lmesh=None*)
Randomly shifts the plane along its axes, enforcing periodic boundary conditions

Parameters

- **seed** (*int.*) – random seed with which to initialize the generator
- **lmesh** (*array*) – the FFT frequency meshgrid (lx,ly) necessary for the calculations in fourier space; if None, a new one is computed from scratch (must have the appropriate dimensions)

save(*filename*, *format=None*, *double_precision=False*)
Saves the Plane to an external file, of which the format can be specified (only fits implemented so far)

Parameters

- **filename** (*str.*) – name of the file on which to save the plane
- **format** (*str.*) – format of the file, only FITS implemented so far; if None, it's detected automatically from the filename
- **double_precision** (*bool.*) – if True saves the Plane in double precision

toFourier()
Switches to Fourier space

toReal()
Switches to real space

class lenstools.simulations.**DensityPlane**(*data*, *angle*, *redshift=2.0*, *cosmology=None*, *comoving_distance=None*, *unit=Unit("rad2")*, *num_particles=None*, *masked=False*, *filename=None*)
Class handler of a lens density plane, inherits from the parent Plane class; additionally it defines redshift and comoving distance attributes that are needed for ray tracing operations

potential(*lmesh=None*)
Computes the lensing potential from the density plane solving the Poisson equation via FFTs

Parameters **lmesh** (*array*) – the FFT frequency meshgrid (lx,ly) necessary for the calculations in fourier space; if None, a new one is computed from scratch (must have the appropriate dimensions)

Returns PotentialPlane instance with the computed lensing potential

class lenstools.simulations.**PotentialPlane**(*data*, *angle*, *redshift=2.0*, *cosmology=None*, *comoving_distance=None*, *unit=Unit("rad2")*, *num_particles=None*, *masked=False*, *filename=None*)
Class handler of a lens potential plane, inherits from the parent Plane class; additionally it defines redshift and comoving distance attributes that are needed for ray tracing operations

deflectionAngles(*x=None*, *y=None*, *lmesh=None*)
Computes the deflection angles for the given lensing potential by taking the gradient of the potential map; it is also possible to proceed with FFTs

Parameters

- **x** (*array with units*) – optional; if not None, compute the deflection angles only for rays hitting the lens at the particular x positions (mainly for speedup in case there are less light rays than the plane resolution allows; must proceed in real space to allow speedup)

- **y** (*array with units*) – optional; if not None, compute the deflection angles only for rays hitting the lens at the particular y positions (mainly for speedup in case there are less light rays than the plane resolution allows; must proceed in real space to allow speedup)
- **lmesh** (*array*) – the FFT frequency meshgrid (lx,ly) necessary for the calculations in fourier space; if None, a new one is computed from scratch (must have the appropriate dimensions)

Returns DeflectionPlane instance, or array with deflections of rays hitting the lens at (x,y)

density(*x=None*, *y=None*)

Computes the projected density fluctuation by taking the laplacian of the potential; useful to check if the potential is reasonable

Parameters

- **x** (*array with units*) – optional; if not None, compute the density only for rays hitting the lens at the particular x positions (mainly for speedup in case there are less light rays than the plane resolution allows; must proceed in real space to allow speedup)
- **y** (*array with units*) – optional; if not None, compute the density only for rays hitting the lens at the particular y positions (mainly for speedup in case there are less light rays than the plane resolution allows; must proceed in real space to allow speedup)

Returns DensityPlane instance with the density fluctuation data (if x and y are None), or numpy array with the same shape as x and y

shearMatrix(*x=None*, *y=None*, *lmesh=None*)

Computes the shear matrix for the given lensing potential; it is also possible to proceed with FFTs

Parameters

- **x** (*array with units*) – optional; if not None, compute the shear matrix only for rays hitting the lens at the particular x positions (mainly for speedup in case there are less light rays than the plane resolution allows; must proceed in real space to allow speedup)
- **y** (*array with units*) – optional; if not None, compute the shear matrix only for rays hitting the lens at the particular y positions (mainly for speedup in case there are less light rays than the plane resolution allows; must proceed in real space to allow speedup)
- **lmesh** (*array*) – the FFT frequency meshgrid (lx,ly) necessary for the calculations in fourier space; if None, a new one is computed from scratch (must have the appropriate dimensions)

Returns ShearTensorPlane instance, or array with deflections of rays hitting the lens at (x,y)

class lenstools.simulations.raytracing.**DeflectionPlane**(*data*, *angle*, *redshift=2.0*, *comoving_distance=None*, *cosmology=None*, *unit=Unit("rad")*)

Class handler of a lens deflection plane, inherits from the parent Spin1 class and holds the values of the deflection angles of the light rays that cross a potential plane

convergence(*precision='first'*)

Computes the convergence from the deflection angles by taking the appropriate components of the jacobian

Parameters **precision** (*str.*) – if "first" computes the convergence at first order in the lensing potential (only one implemented so far)

Returns ConvergenceMap instance with the computed convergence values

jacobian()
Computes the jacobian of the deflection angles, useful to compute shear and convergence; units are handled properly

Returns the jacobian of the deflection field in array form, of shape (4,:,:) where the four components are, respectively, xx,xy,yx,yy

omega(*precision='first'*)
Computes the omega field (i.e. the real space B mode) from the deflection angles by taking the appropriate components of the jacobian

Parameters **precision** (*str.*) – if "first" computes omega at first order in the lensing potential (only one implemented so far)

Returns Spin0 instance with the computed omega values

shear(*precision='first'*)
Computes the shear from the deflection angles by taking the appropriate components of the jacobian

Parameters **precision** (*str.*) – if "first" computes the shear at first order in the lensing potential (only one implemented so far)

Returns ShearMap instance with the computed convergence values

class lenstools.simulations.raytracing.**ShearTensorPlane**(*data*, *angle*, *redshift=2.0*, *comoving_distance=None*, *cosmology=None*, *unit=Unit(dimensionless)*)
Class handler of a plane of shear matrices, inherits from the parent Spin2 class and holds the 3 values of the symmetric shear matrix (2 diagonal + 1 off diagonal), for each pixel

class lenstools.simulations.**RayTracer**(*lens_mesh_size=None*)
Class handler of ray tracing operations: it mainly computes the path corrections of light rays that travel through a set of gravitational lenses

addLens(*lens_specification*)
Adds a gravitational lens to the ray tracer, either by putting in a lens plane, or by specifying the name of a file which contains the lens specifications

Parameters **lens_specification** – specifications of the lens to add, either in tuple(filename,distance,redshift) or as a PotentialPlane instance

convergenceDirect(*initial_positions*, *z=2.0*, *save_intermediate=False*)
Computes the convergence directly integrating the lensing density along the unperturbed line of sight

Parameters

- **initial_positions** (*numpy array or quantity*) – initial angular positions of the light ray bucket, according to the observer; if unitless, the positions are assumed to be in radians. initial_positions[0] is x, initial_positions[1] is y
- **z** (*float.*) – redshift of the sources
- **save_intermediate** (*bool.*) – save the intermediate values of the convergence as successive lenses are crossed

Returns convergence values at each of the initial positions

displayRays(*initial_positions*, *z=2.0*, *projection='2d'*, *fig=None*, *ax=None*)
Uses matplotlib to display a visual of the lens system and of the deflection that the light rays which traverse it undergo

param initial_positions: initial angular positions of the light ray bucket, according to the observer; if unitless, the positions are assumed to be in radians. initial_positions[0] is x, initial_positions[1] is y :type initial_positions: numpy array or quantity

Parameters

- **z** (*float. or array*) – redshift of the sources; if an array is passes, a redshift must be specified for each ray, i.e. z.shape==initial_positions.shape[1:]
- **projection** (*str.*) – can be "2d" for the projections of the ray positions along x and y or "3d" for the full 3d view
- **fig** (*matplotlib figure*) – figure object that owns the plot

randomRoll(*seed=None*)
Randomly rolls all the lenses in the system along both axes

Parameters **seed** (*int.*) – random seed with which to initialize the generator

reorderLenses()
Reorders the lenses in the ray tracer according to their comoving distance from the observer

reset()
Resets the RayTracer plotting engine

shoot(*initial_positions*, *z=2.0*, *initial_deflection=None*, *precision='first'*, *kind='positions'*, *save_intermediate=False*, *compute_all_deflections=False*, *callback=None*, ***kwargs*)
Shots a bucket of light rays from the observer to the sources at redshift z (backward ray tracing), through the system of gravitational lenses, and computes the deflection statistics

Parameters

- **initial_positions** (*numpy array or quantity*) – initial angular positions of the light ray bucket, according to the observer; if unitless, the positions are assumed to be in radians. initial_positions[0] is x, initial_positions[1] is y
- **z** (*float. or array*) – redshift of the sources; if an array is passed, a redshift must be specified for each ray, i.e. z.shape==initial_positions.shape[1:]
- **initial_deflection** (*numpy array or quantity*) – if not None, this is the initial deflection light rays undergo with respect to the line of sight (equivalent to specifying the first derivative IC on the lensing ODE); must have the same shape as initial_positions
- **precision** (*str.*) – precision at which to compute weak lensing quantities, must be "first" for first order in the lensing potential, or "second" for added precision (not functional yet)
- **kind** (*str.*) – what deflection statistics to compute; "positions" will calculate the ray deflections after they crossed the last lens, "jacobian" will compute the lensing jacobian matrix after the last lens, "shear" and "convergence" will compute the omonimous weak lensing statistics
- **save_intermediate** (*bool.*) – save the intermediate positions of the rays too
- **compute_all_deflections** (*bool.*) – if True, computes the gradients of the lensing potential at every pixel on the lens(might be overkill if Nrays<<Npixels); must be True if the computation is done with FFTs
- **callback** (*callable*) – if not None, this callback function is called on the current ray positions array at each step in the ray tracing; the current raytracing instance and the step number are passed as additional arguments, hence callback must match this signature
- **kwargs** (*dict.*) – the keyword arguments are passed to the callback if not None

Returns angular positions (or jacobians) of the light rays after the last lens crossing

shootForward(*source_positions*, *z=2.0*, *save_intermediate=False*, *grid_resolution=512*, *interpolation='nearest'*)

Shoots a bucket of light rays from the source at redshift z to the observer at redshift 0 (forward ray tracing) and computes the according deflections using backward ray tracing plus a suitable interpolation scheme (KD Tree based)

Parameters

- **source_positions** (*numpy array or quantity*) – angular positions of the unlensed sources
- **z** (*float.*) – redshift of the sources
- **save_intermediate** (*bool.*) – if True computes and saves the apparent image distortions after each lens is crossed (can be computationally expensive)
- **grid_resolution** (*int.*) – the number of points on a side of the interpolation grid (must be choosen big enough according to the number of sources to resolve)
- **interpolation** (*str.*) – only "nearest" implemented so far

Returns apparent positions of the sources as seen from the observer

9.1.11 Weak Lensing Simulation Pipeline

Directory tree handling

class lenstools.pipeline.simulation.**SimulationBatch**(*environment*, *syshandler=<lenstools.pipeline.remote.LocalSystem object>*, *indicize=False*)

Class handler of a batch of weak lensing simulations that share the same environment settings

archive(*name*, *pool=None*, *chunk_size=1*, ***kwargs*)

Archives a batch available resource to a tar gzipped archive; the resource file/directory names are retrieved with the list method. The archives will be written to the simulation batch storage directory

Parameters

- **name** (*str.*) – name of the archive
- **pool** (*MPIPool*) – MPI Pool used to parallelize compression
- **kwargs** (*dict.*) – the keyword arguments are passed to the list method

available

Lists all currently available models in the home and storage directories

Returns list.

Return type *SimulationModel*

commit(*message*)

If the Simulation Batch is put under version control in a git repository, this method commits the newly added models,collections,realizations or map/plane sets

Parameters **message** (*str.*) – commit message

copyTree(*path*, *syshandler=<lenstools.pipeline.remote.LocalSystem object>*)

Copies the current batch directory tree into a separate path

Parameters

- **path** (*str.*) – path into which to copy the current batch directory tree
- **syshandler** (*SystemHandler*) – system handler (can be a remote)

classmethod current (*name='environment.ini'*, *syshandler=<lenstools.pipeline.remote.LocalSystem object>*, *indicize=False*)

This method looks in the current directory and looks for a configuration file named "environment.ini"; if it finds one, it returns a SimulationBatch instance that corresponds to the one pointed to by "environment.ini" (default)

Parameters

- **name** (*str.*) – name of the INI file with the environment settings, defaults to 'environment.ini'
- **syshandler** (*SystemHandler*) – system handler that allows to override the methods used to create directories and do I/O from files, must implement the abstract type SystemHandler

Returns Simulation batch pointed to by "environment.ini", or None

Return type *SimulationBatch*

getModel (*cosmo_id*)

Instantiate a SimulationModel object corresponding to the cosmo_id provided

Parameters **cosmo_id** (*str.*) – cosmo_id of the model

Return type *SimulationModel*

info

Returns summary info of the simulation batch corresponding to the current environment

Returns info in dictionary format

list (*resource=None*, *which=None*, *chunk_size=10*, ***kwargs*)

Lists the available resources in the simulation batch (collections,mapsets,etc...)

Parameters

- **resource** (*None or callable*) – custom function to call on each batch.models element, must return a string. If None the list of Storage model directories is returned
- **which** (*tuple. or callable*) – extremes of the model numbers to get (if None all models are processed); if callable, filter(which,self.models) gives the models to archive
- **chunk_size** (*int.*) – size of output chunk
- **kwargs** (*dict.*) – the keyword arguments are passed to resource

Returns requested resources

Return type list.

newModel (*cosmology*, *parameters*)

Create a new simulation model, given a set of cosmological parameters

Parameters

- **cosmology** (*LensToolsCosmology*) – cosmological model to simulate
- **parameters** (*list.*) – cosmological parameters to keep track of

Return type *SimulationModel*

unpack(*where*, *which=None*, *pool=None*)
Unpacks the compressed simulation batch products into the storage directory: the resources of each model must be contained in a file called <cosmo_id>.tar.gz

Parameters

- **where** (*str.*) – path of the compressed resources
- **which** (*tuple.*) – extremes of the model numbers to unpack (if None all models are unpacked)
- **pool** (*MPIPool*) – MPI Pool used to parallelize de-compression

writeCAMBSubmission(*realization_list*, *job_settings*, *job_handler*, *chunks=1*, ***kwargs*)
Writes CAMB submission script

Parameters

- **realization_list** (*list. of str.*) – list of ics to generate in the form "cosmo_id|geometry_id"
- **chunks** (*int.*) – number of independent jobs in which to split the submission (one script per job will be written)
- **job_settings** (JobSettings) – settings for the job (resources, etc...)
- **job_handler** (JobHandler) – handler of the cluster specific features (job scheduler, architecture, etc...)
- **kwargs** (*dict.*) – you can set one_script=True to include all the executables sequentially in a single script

writeNGenICSubmission(*realization_list*, *job_settings*, *job_handler*, *chunks=1*, ***kwargs*)
Writes NGenIC submission script

Parameters

- **realization_list** (*list. of str.*) – list of ics to generate in the form "cosmo_id|geometry_id|icN"
- **chunks** (*int.*) – number of independent jobs in which to split the submission (one script per job will be written)
- **job_settings** (JobSettings) – settings for the job (resources, etc...)
- **job_handler** (JobHandler) – handler of the cluster specific features (job scheduler, architecture, etc...)
- **kwargs** (*dict.*) – you can set one_script=True to include all the executables sequentially in a single script

writeNbodySubmission(*realization_list*, *job_settings*, *job_handler*, *chunks=1*, ***kwargs*)
Writes N-body simulation submission script

Parameters

- **realization_list** (*list. of str.*) – list of ics to generate in the form "cosmo_id|geometry_id|icN"
- **chunks** (*int.*) – number of independent jobs in which to split the submission (one script per job will be written)
- **job_settings** (JobSettings) – settings for the job (resources, etc...)
- **job_handler** (JobHandler) – handler of the cluster specific features (job scheduler, architecture, etc...)

- **kwargs** (*dict.*) – you can set one_script=True to include all the executables sequentially in a single script

writePlaneSubmission(*realization_list*, *job_settings*, *job_handler*, *chunks=1*, ***kwargs*)
Writes lens plane generation submission script

Parameters

- **realization_list** (*list. of str.*) – list of ics to generate in the form "cosmo_id|geometry_id|icN"
- **chunks** (*int.*) – number of independent jobs in which to split the submission (one script per job will be written)
- **job_settings** (JobSettings) – settings for the job (resources, etc...)
- **job_handler** (JobHandler) – handler of the cluster specific features (job scheduler, architecture, etc...)
- **kwargs** (*dict.*) – keyword arguments accepted are "environment_file" to specify the environment settings for the current batch, "plane_config_file" to specify the lensing option for plane generation script. Additionally you can set one_script=True to include all the executables sequentially in a single script

writeRaySubmission(*realization_list*, *job_settings*, *job_handler*, *chunks=1*, ***kwargs*)
Writes raytracing submission script

Parameters

- **realization_list** (*list. of str.*) – list of ics to generate in the form "cosmo_id|geometry_id"
- **chunks** (*int.*) – number of independent jobs in which to split the submission (one script per job will be written)
- **job_settings** (JobSettings) – settings for the job (resources, etc...)
- **job_handler** (JobHandler) – handler of the cluster specific features (job scheduler, architecture, etc...)
- **kwargs** (*dict.*) – keyword arguments accepted are "environment_file" to specify the environment settings for the current batch, "raytracing_config_file" to specify the lensing option for the ray tracing. Additionally you can set one_script=True to include all the executables sequentially in a single script

writeSubmission(*realization_list*, *job_settings*, *job_handler*, *job_executable='lenstools.custom'*, *chunks=1*, ***kwargs*)
Writes a generic job submission script

Parameters

- **realization_list** (*list. of str.*) – list of ics to generate in the form "cosmo_id|geometry_id"
- **chunks** (*int.*) – number of independent jobs in which to split the submission (one script per job will be written)
- **job_settings** (JobSettings) – settings for the job (resources, etc...)
- **job_handler** (JobHandler) – handler of the cluster specific features (job scheduler, architecture, etc...)
- **job_executable** (*str.*) – name of the executable that will be run

- **kwargs** (*dict.*) – keyword arguments accepted are "environment_file" to specify the environment settings for the current batch, "config_file" to specify the job specific configuration file. Additionally you can set one_script=True to include all the executables sequentially in a single script

class lenstools.pipeline.simulation.**SimulationModel**(*cosmology=LensToolsCosmology(H0=72 km / (Mpc s), Om0=0.26, Ode0=0.74, sigma8=0.8, ns=0.96, w0=-1, wa=0, Tcmb0=2.725 K, Neff=3.04, m_nu=[0. 0. 0.] eV, Ob0=0.046), environment=None, parameters=['Om', 'Ol', 'w', 'ns', 'si'], **kwargs*)

Class handler of a weak lensing simulation model, defined by a set of cosmological parameters

collections

Lists all the available collections for a model

Returns list.

Return type *SimulationCollection*

getTelescopicMapSet(*setname*)

Return an instance of the telescopic map set named "name"

Parameters **setname** (*str.*) – name of the map set

Return type *SimulationTelescopicMaps*

mkdir(*directory*)

Create a sub-directory inside the current instance home and storage paths

Parameters **directory** (*str.*) – name of the directory to create

newCollection(*box_size=<Quantity 240.0 Mpc>*, *nside=512*)

Instantiate new simulation with the specified settings

newTelescopicMapSet(*collections*, *redshifts*, *settings*)

Create a new telescopic map set with the provided settings

Parameters

- **collections** (*list.*) – list of the SimulationCollection instances that participate in the telescopic map set
- **redshifts** (*array.*) – redshifts that mark the transition from one collection to the other during raytracing
- **settings** (TelescopicMapSettings) – settings for the new map set

Return type *SimulationMaps*

path(*filename*, *where='storage_subdir'*)

Returns the complete path to the resource corresponding to filename; returns None if the resource does not exist

Parameters

- **filename** (*str.*) – name of the resource
- **where** (*str.*) – where to look for the resource, can be "home_subdir" or "storage_subdir"

Returns full path to the resource

Return type str.

telescopicmapsets
Lists all the available telescopic map sets for a model

Returns list.

Return type *SimulationTelescopicMaps*

class lenstools.pipeline.simulation.**SimulationCollection**(*cosmology*, *environment*, *parameters*, *box_size*, *nside*, ***kwargs*)
Class handler of a collection of simulations that share model parameters

camb2ngenic(*z*)
Read CAMB power spectrum file and convert it in a N-GenIC readable format

Parameters **z** (*float.*) – redshift of the matter power spectrum file to convert

getMapSet(*setname*)
Get access to a pre-existing map set

Parameters **setname** (*str.*) – name of the map set to access

newMapSet(*settings*)
Create a new map set with the provided settings

Parameters **settings** (MapSettings) – settings for the new map set

Return type *SimulationMaps*

realizations
List the available realizations (or independent simulations) for the current collection

Returns list.

Return type *SimulationIC*

writeCAMB(*z*, *settings*)
Generates the parameter file that CAMB needs to read to evolve the current initial condition in time

Parameters

- **settings** (CAMBSettings) – CAMB tunable settings
- **z** (*array.*) – redshifts at which CAMB needs to compute the matter power spectrum

class lenstools.pipeline.simulation.**SimulationIC**(*cosmology*, *environment*, *parameters*, *box_size*, *nside*, *ic_index*, *seed*, *ICFilebase='ics'*, *SnapshotFileBase='snapshot'*, ***kwargs*)
Class handler of a simulation with a defined initial condition

getPlaneSet(*setname*)
Instantiate a SimulationPlanes object that handles a specific plane set; returns None if the plane set does not exist

Parameters **setname** (*str.*) – name of the plane set

Return type *SimulationPlanes*

writeGadget2(*settings*)
Generates the parameter file that Gadget2 needs to read to evolve the current initial condition in time

Parameters **settings** (Gadget2Settings) – Gadget2 tunable settings

writeNGenIC(*settings*)
Generates the parameter file that NGenIC needs to read to generate the current initial condition

Parameters **settings** (NGenICSettings) – NGenIC tunable settings

class lenstools.pipeline.simulation.**SimulationPlanes**(*cosmology*, *environment*, *parameters*, *box_size*, *nside*, *ic_index*, *seed*, *ICFilebase*, *SnapshotFileBase*, *settings*, ***kwargs*)
Class handler of a set of lens planes belonging to a particular simulation

class lenstools.pipeline.simulation.**SimulationMaps**(*cosmology*, *environment*, *parameters*, *box_size*, *nside*, *settings*, ***kwargs*)
Class handler of a set of lensing maps, which are the final products of the lensing pipeline

class lenstools.pipeline.simulation.**SimulationTelescopicMaps**(*cosmology*, *environment*, *parameters*, *collections*, *redshifts*, *settings*, ***kwargs*)
Class handler of a set of lensing telescopic maps, which are the final products of the lensing pipeline

class lenstools.pipeline.simulation.**SimulationCatalog**(*cosmology*, *environment*, *parameters*, *box_size*, *nside*, *settings*, ***kwargs*)
Class handler of a simulated lensing catalog, which is the final products of the lensing pipeline

Tunable settings

class lenstools.pipeline.settings.**EnvironmentSettings**(*home='SimTest/Home'*, *storage='SimTest/Storage'*)
This class handles the system specific environment settings, such as directory paths, modules, etc...

class lenstools.simulations.camb.**CAMBSettings**(***kwargs*)

write(*output_root*, *cosmology*, *redshifts*)
Writes a CAMB parameter file

Parameters

- **output_root** (*str.*) – output_root for the files that CAMB will produce in output
- **cosmology** (*FLRW*) – cosmological model to generate the parameter file for
- **redshifts** (*array.*) – redshifts on which to compute the matter power spectrum and transfer function

Returns string object

Return type StringIO

class lenstools.pipeline.settings.**NGenICSettings**(***kwargs*)
Class handler of NGenIC settings

class lenstools.simulations.**Gadget2Settings**(***kwargs*)
Class handler of the tunable settings in a Gadget2 run

classmethod default()
Generate default settings

writeSection(*section*)
Writes the corresponding section of the Gadget2 parameter file

class lenstools.pipeline.settings.**PlaneSettings**(***kwargs*)
Class handler of plane generation settings

class lenstools.pipeline.settings.**MapSettings**(***kwargs*)
Class handler of map generation settings

class lenstools.pipeline.settings.**TelescopicMapSettings**(***kwargs*)
Class handler of telescopic simulation map generation settings

class lenstools.pipeline.settings.**CatalogSettings**(***kwargs*)
Class handler of simulated catalog generation settings

class lenstools.pipeline.settings.**JobSettings**(***kwargs*)
Class handler of batch job submission settings

Cluster deployment

class lenstools.pipeline.deploy.**JobHandler**

writeExecution(*executables*, *cores*, *settings*)
Write the execution part of the script

Parameters

- **executables** (*list.*) – list of executables to run on the compute nodes
- **cores** (*list.*) – list of numbers of cores for each executable (must have the same length as executables)
- **settings** (JobSettings) – job settings

Returns StringIO object

writePreamble(*settings*, *auto_num_nodes=True*)
Writes the preamble of the job script (resources request,job name, etc...)

Parameters

- **settings** (JobSettings) – job settings
- **auto_num_nodes** (*bool.*) – if True, the number of requested nodes is computed automatically from the number of requested cores (knowing the cluster specifications)

Returns StringIO object

class lenstools.pipeline.deploy.**Directives**(***kwargs*)

class lenstools.pipeline.deploy.**ClusterSpecs**(***kwargs*)

Cluster specific settings

class lenstools.pipeline.cluster.**CoriHandler**
Job handler for the NERSC Cori Phase 1 cluster

writeExecution(*executables*, *cores*, *settings*)
Write the execution part of the script

Parameters

- **executables** (*list.*) – list of executables to run on the compute nodes

- **cores** (*list.*) – list of numbers of cores for each executable (must have the same length as executables)
- **settings** (JobSettings) – job settings

Returns StringIO object

writePreamble(*settings*, *auto_num_nodes=True*)
Writes the preamble of the job script (resources request,job name, etc...)

Parameters

- **settings** (JobSettings) – job settings
- **auto_num_nodes** (*bool.*) – if True, the number of requested nodes is computed automatically from the number of requested cores (knowing the cluster specifications)

Returns StringIO object

class lenstools.pipeline.cluster.**EdisonHandler**
Job handler for the NERSC Edison cluster

writeExecution(*executables*, *cores*, *settings*)
Write the execution part of the script

Parameters

- **executables** (*list.*) – list of executables to run on the compute nodes
- **cores** (*list.*) – list of numbers of cores for each executable (must have the same length as executables)
- **settings** (JobSettings) – job settings

Returns StringIO object

writePreamble(*settings*, *auto_num_nodes=True*)
Writes the preamble of the job script (resources request,job name, etc...)

Parameters

- **settings** (JobSettings) – job settings
- **auto_num_nodes** (*bool.*) – if True, the number of requested nodes is computed automatically from the number of requested cores (knowing the cluster specifications)

Returns StringIO object

class lenstools.pipeline.cluster.**StampedeHandler**
Job handler for the TACC Stampede cluster

writeExecution(*executables*, *cores*, *settings*)
Write the execution part of the script

Parameters

- **executables** (*list.*) – list of executables to run on the compute nodes
- **cores** (*list.*) – list of numbers of cores for each executable (must have the same length as executables)
- **settings** (JobSettings) – job settings

Returns StringIO object

writePreamble(*settings*, *auto_num_nodes=True*)
Writes the preamble of the job script (resources request,job name, etc...)

Parameters

- **settings** (JobSettings) – job settings
- **auto_num_nodes** (*bool.*) – if True, the number of requested nodes is computed automatically from the number of requested cores (knowing the cluster specifications)

Returns StringIO object

9.1.12 Real observation sets

class lenstools.observations.**CFHTLens**(*root_path=None*)

Class handler of the CFHTLens reduced data set, already split in 13 3x3 deg^2 subfields

getName(*subfield=1*, *smoothing=0.5*)

Get the names of the FITS files where the subfield images are stored

Parameters

- **subfield** (*int.*) – the subfield number (1-13)
- **smoothing** (*float.*) – the smoothing scale of the subfield image, in arcminutes

Returns str. : the FITS file name

load(***kwargs*)

Loads in a CFHT observation as a ConvergenceMap instance

Parameters **kwargs** – the keyword arguments are passed to the getName method

Returns ConvergenceMap instance

9.1.13 Limber integration

class lenstools.simulations.limber.**LimberIntegrator**(*cosmoModel=FlatLambdaCDM(name="WMAP9", H0=69.3 km / (Mpc s), Om0=0.286, Tcmb0=2.725 K, Neff=3.04, m_nu=[0. 0. 0.] eV, Ob0=0.0463)*)

A 3D power spectrum integrator that will compute the convergence power spectrum using the Limber approximation.

Parameters **cosmoModel** (*astropy.cosmology*) – One of astropy.cosmology objects (WMAP9 cosmology is set by default)

convergencePowerSpectrum(*l*)

Computes the convergence power spectrum with the Limber integral of the 3D matter power spectrum; this still assumes a single source redshift at z0 = max(z)

Parameters **l** (*array*) – multipole moments at which to compute the convergence power spectrum

Returns array – the convergence power spectrum at the l values specified

load3DPowerSpectrum(*loader*, **args*, ***kwargs*)

Loads in the matter power spectrum from (pre-computed) external files; args and kwargs are passed to the loader

Parameters **loader** (*function*) – must return, in order, k, z, P(k,z)

setUnits(*kappa_units=None*, *power_units=None*)

Set the physical units for wavenumber and matter power spectrum, default for length is Mpc

9.1.14 Defaults

lenstools.utils.defaults.**load_fits_default_convergence**(*filename*)
This is the default convergence fits file loader, it assumes that the two components of the shear are stored in two different image FITS files, which have an ANGLE keyword in the header

Parameters **gamma_file** – Name of the FITS file that contains the shear map

Returns tuple – (angle,ndarray – kappa; kappa is the convergence map)

Raises IOError if the FITS files cannot be opened or do not exist

lenstools.utils.defaults.**load_fits_default_shear**(*filename*)
This is the default shear fits file loader, it assumes that the two components of the shear are stored in a single image FITS file, which have an ANGLE keyword in the header

Parameters **gamma_file** – Name of the FITS file that contains the shear map

Returns tuple – (angle,ndarray – gamma; gamma[0] is the gamma1 map, gamma[1] is the gamma2 map); the maps must follow matrix ordering, i.e. the first axis (0) is y and the second axis (1) is x. This matters for the E/B mode decomposition

Raises IOError if the FITS files cannot be opened or do not exist

lenstools.utils.defaults.**measure_power_spectrum**(*filename*, *l_edges*, *columns=None*)
Default ensemble loader: reads a FITS data file containing a convergence map and measures its power spectrum

Parameters **args** (*Dictionary*) – A dictionary that contains all the relevant parameters as keys. Must have a "map_id" key

Returns ndarray of the measured statistics

Raises AssertionError if the input dictionary doesn't have the required keywords

9.1.15 Miscellaneous utilities

MPI

class lenstools.utils.mpi.**MPIWhirlPool**(*comm=None*, *debug=False*, *loadbalance=False*)
MPI class handler, inherits from MPI pool and adds one sided communications utilities (using RMA windows)

accumulate(*op=None*)
Accumulates the all the window data on the master, performing a custom operation (default is sum)

bcast(**args*, ***kwargs*)
Equivalent to mpi4py `bcast()` collective operation.

close()
Just send a message off to all the pool members which contains the special `_close_pool_message` sentinel.

closeWindow()
Closes a previously opened RMA window

get(*process*)
Read data from an RMA window open on a particular process

is_master()
Is the current process the master?

map(*function*, *tasks*)
Like the built-in `map()` function, apply a function to all of the values in a list and return the list of results.

Parameters

- **function** – The function to apply to the list.
- **tasks** – The list of elements.

openWindow(*memory*, *window_type='sendrecv'*)
Create a RMA window that looks from the master process onto all the other workers

Parameters **memory** (*numpy nd array*) – memory buffer on which to open the window

wait()
If this isn't the master process, wait for instructions.

Fast Fourier Transforms

class lenstools.utils.fft.**NUMPYFFTPack**

Algorithms

class lenstools.utils.algorithms.**pcaHandler**(*constructor_series*, *constructor_ensemble*, *columns*, *location*, *scale*)
Handles principal component analysis

Decorators

CHAPTER 10

Issues

The code is maintained and developed on github, pull requests are welcome!

CHAPTER 11

License

Copyright 2014 Andrea Petri and contributors; lenstools is free software made available under the MIT License. For details see the LICENSE file

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

d

l

o

s

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W