Pylians3

Release 1.0

Francisco Villaescusa-Navarro

Aug 25, 2023

INSTALLATION

Pylians stands for **Py**thon **li**braries for the **a**nalysis of **n**umerical **s**imulations. They are a set of python libraries, written in python, cython and C, whose purposes is to facilitate the analysis of numerical simulations (both N-body and hydrodynamic). Pylians runs on both python2 and python3. Among other things, they can be used to:

- Compute density fields
- Compute power spectra
- Compute bispectra
- Compute correlation functions
- Identify voids
- Populate halos with galaxies using an HOD
- Apply HI+H2 corrections to the output of hydrodynamic simulations
- Make 21cm maps
- Compute DLAs column density distribution functions
- Plot density fields and make movies

Pylians were the native or inhabitant of the Homeric town of Pylos.

CHAPTER

ONE

INSTALLATION

Linux

Stable

```
python -m pip install Pylians
```

Development

```
git clone https://github.com/franciscovillaescusa/Pylians3.git
cd Pylians3
python -m pip install .
```

To verify that the installation was successful do

```
python Tests/import_libraries.py
```

If no output is produced, everything went fine.

Note: You may need to add specific compilation flags for your system if the above procedure fails. For instance, for a Power9 system such as CINECA Marconi 100 you need to add these compilations flags to `extra_compilation_args`: `'-mcpu=powerpc64le'` and `'-mtune=powerpc64le'`.

Mac

Stable

```
python -m pip install Pylians
```

If this does not work on your machine try to install the development version.

Development

The main problem installing Pylians on a mac is due to openmp. So these instructions will remove that functionality.

1. Download Pylians3 code from the repository:

   ```
   git clone https://github.com/franciscovillaescusa/Pylians3.git
   ```

2. Create a new conda environment and activate it

   ```
   conda create --name Pylians_env python=3.x.x
   conda activate Pylians_env
   ```

   3.x.x should be changed with your python version, e.g. 3.9.7

3. Install llvm and libomp using homebrew

```
brew install llvm
brew install libomp
```

4. Navigate to the Pylians3 directory. Open `setup.py` in a text editor. Delete every instance of `-fopenmp`
5. In a terminal, while in Pylians3, type

```
python -m pip install .
```

6. Test the installation by executing

```
python Tests/import_libraries.py
```

   If no output is produced, everything went fine.

These instructions have been tested on High Sierra. Many thanks to Alexander Gough for this!

Note: For M1 mac users the instructions are similar but:

- verify that clang version is >=13
- replace `-march=native` by `mcpu=apple-m1`
- execute `CC=clang python setup.py install`

Thanks for Valerio Marra for this!

Note: Pylians works for both python2 and python3. Depending on your python version, it will install accordingly. Sometimes, python3 needs to be invoked explicitly as `python3`. If so, install as `python3 -m pip install Pylians`

Warning: When facing problems installing Pylians due to openmp, its functionality can be disabled by removing the -fopenmp flags in the `setup.py` file (see Instructions for Mac development version). This requires the installation in the development mode.

1.1 Dependencies

Pylians make use of these packages

- `numpy`
- `scipy`
- `h5py`
- `pyfftw`
- `cython`

that would be installed automatically when invoking the above pip command. Note that Pylians also requires a working openmp environment. This needs to be installed separately before invoking the pip command. If there are conflicts with old versions, try to upgrade to the latest versions of the required packages, e.g.

```
python -m pip install --upgrade numpy
```

1.2 Upgrade

To upgrade to the latest version

Stable

```
python -m pip install --upgrade Pylians
```

Development

```
cd Pylians3
git pull
python -m pip install .
```

In a Mac, if having problems with openmp remove the instances in the `setup.py` file before executing `python -m pip install .`

CHAPTER

TWO

CONSTRUCTION

Pylians provides the routine `MA` to construct 2D and 3D density fields from the positions of particles. That routine can also construct marked fields, by weigthing each particle according to some weigh. The arguments of that routine are these:

- `pos`. The positions of the particles, either in 2D or 3D. It should be a numpy float32 array; e.g. in 3D should be something like `pos = np.zeros((1000,3), dtyp=np.float32)`.
- `field`. This is a numpy float32 array in either 2D or 3D that will contain the density field.
- `BoxSize`. Size of the cubic region (in 3D) or the rectangular plane (2D).
- `MAS`. Mass-assignment scheme used to deposit particles mass to the grid. Options are: `'NGP'` (nearest grid point), `'CIC'` (cloud-in-cell), `'TSC'` (triangular-shape cloud), `'PCS'` (piecewise cubic spline). For most applications `'CIC'` is enough.
- `W`. The weight associated to each particle, if any. If no weights used, set it `None`.
- `verbose`. Whether to print some information on the progress.

Note: If you want to construct a field in redshift-space, you will need the particle positions in redshift-space. See *Redshift-space distortions* on how move particles, halos, galaxies...etc, from real to redshift-space.

We now provide examples on how to use this routine:

2.1 Density field in 3D

This example shows how to compute the density constrast field from the positions of particles in 3D.

```
import numpy as np
import MAS_library as MASL

# number of particles
Np = 128**3

# density field parameters
grid    = 128    #the 3D field will have grid x grid x grid voxels
BoxSize = 1000.0 #Mpc/h ; size of box
MAS     = 'CIC'  #mass-assigment scheme
verbose = True   #print information on progress

# particle positions in 3D
```

(continues on next page)

(continued from previous page)

```
pos = np.random.random((Np,3)).astype(np.float32)*BoxSize

# define 3D density field
delta = np.zeros((grid,grid,grid), dtype=np.float32)

# construct 3D density field
MASL.MA(pos, delta, BoxSize, MAS, verbose=verbose)

# at this point, delta contains the effective number of particles in each voxel
# now compute overdensity and density constrast
delta /= np.mean(delta, dtype=np.float64);  delta -= 1.0
```

After the last line, `delta` contains the density constrast field, defined as $\delta(x) = \rho(x)/\bar{\rho} - 1$, where $\rho(x)$ is the value of the density field at position x.

2.2 Density field in 2D

This example shows how to compute the density constrast field from the positions of particles in 2D.

```
import numpy as np
import MAS_library as MASL

# number of particles
Np = 256**2

# density field parameters
grid    = 256    #the 2D field will have grid x grid pixels
BoxSize = 1000.0 #Mpc/h ; size of box
MAS     = 'TSC'  #mass-assigment scheme
verbose = True   #print information on progress

# particle positions in 2D
pos = np.random.random((Np,2)).astype(np.float32)*BoxSize

# define 2D density field
delta = np.zeros((grid,grid), dtype=np.float32)

# construct 2D density field
MASL.MA(pos, delta, BoxSize, MAS, verbose=verbose)

# at this point, delta contains the effective number of particles in each pixel
# now compute overdensity and density constrast
delta /= np.mean(delta, dtype=np.float64);  delta -= 1.0
```

After the last line, `delta` contains the density constrast field, defined as $\delta(x) = \rho(x)/\bar{\rho} - 1$, where $\rho(x)$ is the value of the density field at position x.

2.3 Gas density field in 3D

This example shows how to construct a gas density field in 3D, where the position of the particles, together with their associated gas masses are used.

```
import numpy as np
import MAS_library as MASL

# number of particles
Np = 128**3

# density field parameters
grid    = 128    #the 3D field will have grid x grid x grid voxels
BoxSize = 1000.0 #Mpc/h ; size of box
MAS     = 'CIC'  #mass-assigment scheme
verbose = True   #print information on progress

# particle positions in 3D
pos = np.random.random((Np,3)).astype(np.float32)*BoxSize

# gas masses of the particles (masses goes from 0 to 1)
mass = np.random.random(Np).astype(np.float32) #Msun/h

# define 3D density field
delta = np.zeros((grid,grid,grid), dtype=np.float32)

# construct 3D density field
MASL.MA(pos, delta, BoxSize, MAS, W=mass, verbose=verbose)

# at this point, delta contains the effective gas mass in each voxel
# now compute overdensity and density constrast
delta /= np.mean(delta, dtype=np.float64);  delta -= 1.0
```

After the last line, `delta` contains the gas density constrast field, defined as $\delta_{\rm g}(x) = \rho_{\rm g}(x)/\bar{\rho}_{\rm g} - 1$, where $\rho_{\rm g}(x)$ is the value of the gas density field at position x.

Note: Marked density fields (see e.g. this paper) can be constructed by using the considered mark as a weigh for every particle or galaxy.

CHAPTER

THREE

SMOOTHING

Pylians provides routines to smooth fields with several filters. The ingredients needed are:

- `field`. This is a 3D float numpy array that contains the input field to be smoothed.
- `BoxSize`. This is the size of the box with the input density field.
- `R`. This is the smoothing scale. This argument only matter for filters operating in configuration space: `Top-Hat` and `Gaussian`. For other filters, set this value to `0`.
- `grid`. This is the grid size of the input field, i.e. `field.shape[0]`.
- `threads`. Number of openmp threads to be used.
- `Filter`. Filter to use. `'Top-Hat'`, `'Gaussian'` or `'Top-Hat-k'`.
- `kmin`. The minimum value of the wavenumber when using the `Top-Hat-k` filter. This argument only need to be specified when using this filter.
- `kmax`. The maximum value of the wavenumber when using the `Top-Hat-k` filter. This argument only need to be specified when using this filter.
- `W_k`. This is a 3D complex64 numpy array containing the Fourier-transform of the filter. Notice that when smoothing a discrete field, like the one stored on a regular grid, the Fourier-transform of the filter need to be computed in the same way as the for the field, i.e. through DFT instead of FT.

An example is this

```
import smoothing_library as SL

BoxSize = 75.0 #Mpc/h
R       = 5.0  #Mpc.h
grid    = field.shape[0]
Filter  = 'Top-Hat'
threads = 28

# compute FFT of the filter
W_k = SL.FT_filter(BoxSize, R, grid, Filter, threads)

# smooth the field
field_smoothed = SL.field_smoothing(field, W_k, threads)
```

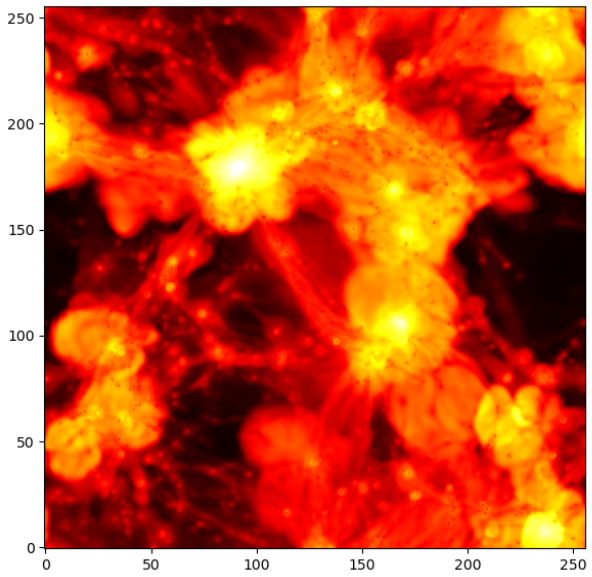
Pylians can also smooth 2D maps by adding `_2D` to the smoothing routines:

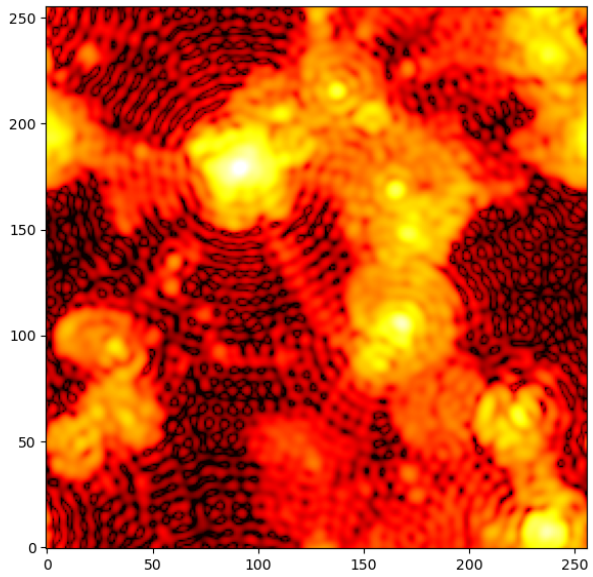
```
import smoothing_library as SL
import numpy as np

```

(continues on next page)

(continued from previous page)

```
# we take as field a temperature image from the CAMELS Multifield Dataset
f_map = 'CMD/2D_maps/data/Maps_T_IllustrisTNG_CV_z=0.00.npy'
field = np.load(f_map)[0] #only take the first image

BoxSize = 25.0 #Mpc/h
grid    = field.shape[0]
R       = 0.0 #only matter for configuration space filters
Filter  = 'Top-Hat-k'
threads = 1
kmin    = 0  #h/Mpc
kmax    = 10 #h/Mpc

# compute the filter in Fourier space
W_k = SL.FT_filter_2D(BoxSize, R, grid, Filter, threads, kmin, kmax)

# smooth the field
field_smoothed = SL.field_smoothing_2D(field, W_k, threads)
```





CHAPTER

FOUR

DENSITY FIELD INTERPOLATION

Sometimes we have a density field and we would like to evalute it at some positions (e.g. particle positions, void positions...etc). Pylians provides the routine `CIC_interp` to accomplish this. That routine uses the Cloud-in-Cell (CIC) scheme to interpolate the density field into the particle positions. The ingredients are:

- `field`. This is the density field. Should be a 3D float numpy array
- `BoxSize`. Size of the density field. Units in Mpc/h
- `pos`. These are the positions of the particles. Units in Mpc/h
- `density_interpolated`. This is a 1D array with the value of the density field at the particle positions. Should be a numpy float array.

An example is this:

```
import MAS_library as MASL

# define the array containing the value of the density field at positions pos
density_interpolated = np.zeros(pos.shape[0], dtype=np.float32)

# find the value of the density field at the positions pos
MASL.CIC_interp(field, BoxSize, pos, density_interpolated)
```

CHAPTER

FIVE

N-BODY SIMULATIONS: GADGET

Pylians provides the routine `density_field_gadget` that simplifies the construction of 3D density fields from Gadget snapshots.

Note: This routine should also work smoothly with AREPO & GIZMO snapshots.

The arguments of this routine are:

- `snapshot`. This is the name of the gadget snapshot. Pylians supports formats 1, 2 and hdf5. Set it as `'snap_001'`, even if the files are `'snap_001.0'`, `'snap_001.1'`, ... or `'snap_001.0.hdf5'`, `'snap_001.1.hdf5'`.
- `grid`. The constructed density field will be a 3D float numpy array with :math: grid*^3`` voxels. The larger this number the higher the resolution, but more memory will be used.
- `ptypes`. Particle type over which compute the density field. It can be individual types, `[0]` (gas), `[1]` (cold dark matter), `[2]` (neutrinos), `[3]` (particle type 3), `[4]` (stars), `[5]` (black holes), or combinations. E.g. `[0,1]` (gas+cold dark matter), `[0,4]` (gas+stars), `[0,1,2,4]` (gas+CDM+neutrinos+stars). For all components (total matter) use `[0,1,2,3,4,5]` or `[-1]`.
- `MAS`. Mass-assignment scheme used to deposit particles mass to the grid. Options are: `'NGP'` (nearest grid point), `'CIC'` (cloud-in-cell), `'TSC'` (triangular-shape cloud), `'PCS'` (piecewise cubic spline). For most applications `'CIC'` is enough.
- `do_RSD`. If `True`, particles positions will be moved to redshift-space along the `axis` axis.
- `axis`. Axis along which redshift-space distortions will be implemented (only needed if `do_RSD=True`): 0, 1 or 2 for x-axis, y-axis or z-axis, respectively.
- `verbose`. Whether to print some information on the routine progress.

This is an example of how to use this routine:

```
import numpy as np
import MAS_library as MASL

snapshot = 'snapdir_010/snap_010'  #snapshot name
grid     = 512                     #grid size
ptypes   = [1,2]                   #CDM + neutrinos
MAS      = 'CIC'                   #Cloud-in-Cell
do_RSD   = False                   #dont do redshif-space distortions
axis     = 0                       #axis along which place RSD; not used here
verbose  = True   #whether print information on the progress
```

(continues on next page)

(continued from previous page)

```
# Compute the effective number of particles/mass in each voxel
delta = MASL.density_field_gadget(snapshot, ptypes, grid, MAS, do_RSD, axis, verbose)

# compute density contrast: delta = rho/<rho> - 1
delta /= np.mean(delta, dtype=np.float64);  delta -= 1.0
```

CHAPTER

SIX

HYDRODYNAMIC SIMULATIONS

6.1 2 dimensions

Imagine that you have a collection of particles at some positions `pos` and each particle represents either a sphere (for SPH simulations) or a voronoi cell (for moving mesh simulations). If you would like to use a more physical mass assignment scheme than NGP, CIC...etc, Pylians provide several routines to deal with these situations and create 2D density fields. We now describe the available routines.

6.1.1 Sampling with tracers

The routine `voronoi_NGP_2D` is designed to take as input particle positions (voronoi cells) in 2D, masses and radii from moving mesh hydrodynamic simulations and compute the density field in a 2D region. This routine works as follows. It considers each particle as a uniform circle and associate the mass on it to the grid itself. It achieves that by splitting the circle into `r_divisions` shells that have the same area. Then, it associates to each shell a number of `particles_per_cell` that are distributed equally in angle. Finally, each of those subparticles belonging to the initial circle, is associated to a grid cell using the NGP mass assignment scheme. Note that this routine can be very computationally expensive if each particle is subsampled with many subparticles. The ingredients needed are:

- `density`. This is the 2D density field that the routine will fill up. It should be a double numpy array.
- `pos`. These are the positions of the particles, either in 2D or 3D. Should be float numpy array.
- `mass`. This is a 1D array with the masses (or other property) of the particles. Should be a float numpy array.
- `radii`. This is a 1D float numpy array with the radii of the particles. If only volume is available, radii can be computed as 4*pi/3*R^3 = Volume.
- `x_min`, `y_min` & `BoxSize`. The routine will compute the density field in a region with coordinates [`x_min:x_min+BoxSize` , `y_min:y_min+BoxSize`]. Units should be Mpc/h.
- `particles_per_cell`. Total number of particles to subsample each particle (voronoi cell).
- `r_divisions`. Number of circular shells to use to subsample each particle (voronoi cell)
- `periodic`. Whether use periodic boundary conditions for the considered region.

```
import numpy as np
import MAS_library as MASL

x_min, y_min, BoxSize = 0.0, 0.0, 25.0 #Mpc/h; origin and size of considered region
grid                  = 512            #size of grid
particles_per_cell = 1000              #total number of tracers to assign to each␣
↪particle
r_divisions           = 7              #number of radial divisions
```

(continues on next page)

(continued from previous page)

```
periodic           = True              #whether the considered region has periodic␣
↪conditions

# define the 2D density field
density = np.zeros((grid,grid), dtype=np.float64)

# compute 2D density field from particle positions, radii and masses
MASL.voronoi_NGP_2D(density, pos, mass, radii, x_min, y_min, BoxSize,
                    particles_per_cell, r_divisions, periodic)
```

6.1.2 Column density: voronoi cells

This routine is designed to take as input particle positions (voronoi cells) in 3D, masses and radii from moving mesh hydrodynamic simulations and compute the column density field in a 2D region. Note that the difference with respect to the above routine is that in this case we compute the projected mass density, not the density itself (as above). This routine works as follows. It considers each particle/cell as a uniform sphere. It then takes a regular grid with the same dimensions as `density`, and in each grid cell computes the projected density of all particles contributing to that line-of-sight. Notice that in this case mass conservation is not fullfilled, as in each grid cell a single line-of-sight is considered. The ingredients needed are:

- `density`. This is the 2D density field that the routine will fill up. It should be a double numpy array.
- `pos`. These are the positions of the particles, either in 2D or 3D. Should be float numpy array.
- `mass`. This is a 1D array with the masses (or other property) of the particles. Should be a float numpy array.
- `radius`. This is a 1D float numpy array with the radii of the particles. If only volume is available, radii can be computed as 4*pi/3*R^3 = Volume.
- `x_min`, `y_min` & `BoxSize`. The routine will compute the density field in a region with coordinates [`x_min:x_min+BoxSize` , `y_min:y_min+BoxSize`]. Units should be Mpc/h.
- `axis_x`, `axis_y`. Integers to select the axes along which made the projection: 0(X), 1(Y) or 2(Z).
- `periodic`. Whether use periodic boundary conditions for the considered region.
- `verbose`. Whether show information on the computation.

```
import numpy as np
import MAS_library as MASL

x_min, y_min, BoxSize = 0.0, 0.0, 25.0 #Mpc/h; origin and size of the considered region
axis_x, axis_y        = 0, 1           #0(X), 1(Y), 2(Z)
grid                  = 512            #grid size
periodic              = True           #whether the considered region has periodic␣
↪conditions

# define the array hosting the 2D field
density = np.zeros((grid,grid), dtype=np.float64)

# compute the density field
MASL.voronoi_RT_2D(density, pos, mass, radius, x_min, y_min,
                   axis_x, axis_y, BoxSize, periodic, verbose=True)
```

Note: More detailed scripts can be found here.

6.1.3 Column density: SPH

This routine is basically the same as the above, but instead of assuming uniform spheres, uses the SPH kernel as its internal density profile. The ingredients needed are:

- `density`. This is the 2D density field that the routine will fill up. It should be a double numpy array.
- `pos`. These are the positions of the particles, either in 2D or 3D. Should be float numpy array.
- `mass`. This is a 1D array with the masses (or other property) of the particles. Should be a float numpy array.
- `radius`. This is a 1D float numpy array with the radii of the particles. If only volume is available, radii can be computed as 4*pi/3*R^3 = Volume.
- `x_min`, `y_min` & `BoxSize`. The routine will compute the density field in a region with coordinates [`x_min:x_min+BoxSize` , `y_min:y_min+BoxSize`]. Units should be Mpc/h.
- `axis_x`, `axis_y`. Integers to select the axes along which made the projection: 0(X), 1(Y) or 2(Z).
- `periodic`. Whether use periodic boundary conditions for the considered region.
- `verbose`. Whether show information on the computation.

```
import numpy as np
import MAS_library as MASL

x_min, y_min, BoxSize = 0.0, 0.0, 25.0 #Mpc/h; origin and size of considered region
axis_x, axis_y        = 0, 1           #0(X), 1(Y), 2(Z)
grid                  = 512            #grid size
periodic              = True           #whether the considered region has periodic␣
↪conditions

# define the array hosting the 2D field
density = np.zeros((grid,grid), dtype=np.float64)

# compute the density field
MASL.SPH_RT_2D(density, pos, mass, radius, x_min, y_min,
               axis_x, axis_y, BoxSize, periodic, verbose=True)
```

6.2 3 dimensions

In hydrodynamic simulations, gas is usually modelled as spheres or voronoi cells. In this case, instead of using the standard mass assignment schemes such as NPG, CIC or TSC, it is better to associate these spheres to the regular grid. We recomment using this code to achieve this:

voxelize

CHAPTER

SEVEN

POWER SPECTRUM

Pylians provide several routines to compute power spectra, that we outline now.

7.1 3D

Pylians provide routines to compute different power spectra for 3 dimensional fields.

7.1.1 Auto-power spectrum

The ingredients needed to compute the auto-power spectra are:

- `delta`. This is the density, overdensity or density contrast field. It should be a 3 dimensional float numpy array such `delta = np.zeros((grid, grid, grid), dtype=np.float32)`. See *Density fields* on how to compute density fields using Pylians.
- `BoxSize`. Size of the periodic box. The units of the output power spectrum depend on this.
- `axis`. Axis along which compute the quadrupole, hexadecapole and the 2D power spectrum. If the field is in real-space set `axis=0`. If the field is in redshift-space set `axis=0`, `axis=1` or `axis=2` if the redshift-space distortions have been placed along the x-axis, y-axis or z-axis, respectively.
- `MAS`. Mass-assignment scheme used to generate the density field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the density field has not been generated with any of these set it to `'None'`. This is used to correct for the MAS when computing the power spectrum.
- `threads`. Number of openmp threads to be used.
- `verbose`. Whether print information on the status/progress of the calculation: True or False

An example on how to compute the power spectrum is this:

```
import Pk_library as PKL

# compute power spectrum
Pk = PKL.Pk(delta, BoxSize, axis, MAS, threads, verbose)

# Pk is a python class containing the 1D, 2D and 3D power spectra, that can be retrieved␣
↪as

# 1D P(k)
k1D      = Pk.k1D
Pk1D     = Pk.Pk1D
```

(continues on next page)

(continued from previous page)

```
Nmodes1D = Pk.Nmodes1D

# 2D P(k)
kpar     = Pk.kpar
kper     = Pk.kper
Pk2D     = Pk.Pk2D
Nmodes2D = Pk.Nmodes2D

# 3D P(k)
k        = Pk.k3D
Pk0      = Pk.Pk[:,0] #monopole
Pk2      = Pk.Pk[:,1] #quadrupole
Pk4      = Pk.Pk[:,2] #hexadecapole
Pkphase  = Pk.Pkphase #power spectrum of the phases
Nmodes   = Pk.Nmodes3D
```

Note: The 1D power spectrum is computed as $P_{1\rm D}(k_\parallel) = \int \frac{d^2\vec{k}_\perp}{(2\pi)^2} P_{3\rm D}(k_\parallel, k_\perp)$. This shouldn't be confused with the traditional 3D power spectrum.

7.1.2 Cross-power spectrum

Pylians also provides routines to compute the auto- and cross-power spectrum of multiple fields. For instance, to compute the auto- and cross-power spectra of two fields, `delta1` and `delta2`:

```
import Pk_library as PKL

Pk = PKL.XPk([delta1,delta2], BoxSize, axis, MAS=['CIC','CIC'], threads=1)
```

A description of the variables `BoxSize`, `axis`, `MAS` and `threads` can be found in *Auto-power spectrum*. As with the auto-power spectrum, `delta1` and `delta2` need to be 3D float numpy arrays. `Pk` is a python class that contains all the following information

```
# 1D P(k)
k1D      = Pk.k1D
Pk1D_1   = Pk.Pk1D[:,0]  #field 1
Pk1D_2   = Pk.Pk1D[:,1]  #field 2
Pk1D_X   = Pk.PkX1D[:,0] #field 1 - field 2 cross 1D P(k)
Nmodes1D = Pk.Nmodes1D

# 2D P(k)
kpar     = Pk.kpar
kper     = Pk.kper
Pk2D_1   = Pk.Pk2D[:,0]  #2D P(k) of field 1
Pk2D_2   = Pk.Pk2D[:,1]  #2D P(k) of field 2
Pk2D_X   = Pk.PkX2D[:,0] #2D cross-P(k) of fields 1 and 2
Nmodes2D = Pk.Nmodes2D

# 3D P(k)
k      = Pk.k3D
```

(continues on next page)

(continued from previous page)

```
Pk0_1  = Pk.Pk[:,0,0]  #monopole of field 1
Pk0_2  = Pk.Pk[:,0,1]  #monopole of field 2
Pk2_1  = Pk.Pk[:,1,0]  #quadrupole of field 1
Pk2_2  = Pk.Pk[:,1,1]  #quadrupole of field 2
Pk4_1  = Pk.Pk[:,2,0]  #hexadecapole of field 1
Pk4_2  = Pk.Pk[:,2,1]  #hexadecapole of field 2
Pk0_X  = Pk.XPk[:,0,0] #monopole of 1-2 cross P(k)
Pk2_X  = Pk.XPk[:,1,0] #quadrupole of 1-2 cross P(k)
Pk4_X  = Pk.XPk[:,2,0] #hexadecapole of 1-2 cross P(k)
Nmodes = Pk.Nmodes3D
```

The `XPk` function can be used for more than two fields, e.g.

```
BoxSize = 1000.0 #Mpc/h
axis    = 0
MAS     = ['CIC','NGP','TSC','None']
threads = 16

Pk = PKL.XPk([delta1,delta2,delta3,delta4], BoxSize, axis, MAS, threads)
```

7.1.3 Gadget snapshots

Pylians provides the routine `Pk_Gadget` that simplifies the computation of auto-power spectra from Gadget snapshots. The arguments of that routine are these:

- `snapshot`. The name of the Gadget snapshot (supports format I, II and hdf5 files). If you have multiple files per snapshot, just use the prefix. For instance, if you have files as `snapdir_004/snap_004.0.hdf5`, `snapdir_004/snap_004.1.hdf5`, `snapdir_004/snap_004.2.hdf5`...etc, use `snapdir_004/snap_004`. For single files, you can use either the prefix or the full name.
- `grid`. The routine will compute the density field on a regular grid with grid x grid x grid voxels. This will basically determine the size of the Nyquist frequency in the power spectrum calculation.
- `particle_type`. The particle types to be used; this routine supports several types. For instance [1] for dark matter, [2] for neutrinos, [4] for stars. It can also be several of them, e.g. [1,2] for dark matter + neutrinos.
- `do_RSD`. Whether move particles to redshift-space and compute power spectrum in redshift-space.
- `axis`. Axis along which place the redshift-space distortions. Only matters if `do_RSD = True`.
- `cpus`. Number of openmp threads to be used in the calculation.
- `folder_out`. Folder where to write the results. If set to `None`, results will be written in the current folder.

An example of how to use this routine is this:

```
import numpy as np
import Pk_library as PKL

# parameters
snapshot      = '/home/Paco/Quijote/Snapshots/fiducial/34/snapdir_004/snap_004'
↪#snapshot name
grid          = 512    #grid size
particle_type = [1]    #use dark matter [1]
do_RSD        = True   #move particles to redshift-space and calculate Pk in redshift-
```

(continues on next page)

(continued from previous page)

```
↪space
axis          = 1      #RSD placed along the y-axis
cpus          = 8      #number of openmp threads
folder_out    = '/home/Paco/Quijote/Pk/fiducial/34' #folder where to write results

# compute power spectrum of the snapshot
PKL.Pk_Gadget(snapshot, grid, particle_type, do_RSD, axis, cpus, folder_out)
```

Calling the routine will compute the auto-power spectrum of the different particle types and their cross-power spectra (for multiple particle types). It will write files for the different auto- and cross-power spectra. The format of the files will be `k Pk0 Pk2 Pk4 Nmodes`, where `k` is the wavenumber in units of h/Mpc (if snapshot is in kpc/h units), `Pk0`, `Pk2`, and `Pk4` are the monopole, quadrupole, and hexadecapole in units of (Mpc/h)^3 and `Nmodes` is the number of modes inside each k-bin.

7.1.4 Marked-power spectrum

The above routines can be used for standard fields or for marked fields. The script below shows an example of how to compute a marked power spectrum where each particle is weighted by its mean density within a radius of 10 Mpc/h (see *Smoothing* to see how to smooth a field).

```
import numpy as np
import MAS_library as MASL
import Pk_library as PKL
import smoothing_library as SL

################################ INPUT ##########################################
# parameters to construct density field
grid    = 512    #grid size
BoxSize = 1000   #Mpc/h
MAS     = 'CIC'  #Cloud-in-Cell

# parameters to smooth the field
R       = 10.0      #Mpc/h; smoothing scale
Filter  = 'Top-Hat' #filter
threads = 1         #openmp threads

# Pk parameters
do_RSD  = False   #whether do redshift-space distortions
axis    = 0       #axis along which place RSD
verbose = True    #whether to print some information on the calculation progress
#################################################################################


###### compute density field #######
# define the array hosting the density constrast field
delta = np.zeros((grid,grid,grid), dtype=np.float32)

# read the particle positions
pos = np.loadtxt('myfile.txt') #Mpc/h
pos = pos.astype(np.float32)   #pos should be a numpy float array
```

(continues on next page)

(continued from previous page)

```
# compute number of particles in each voxel
MASL.MA(pos,delta,BoxSize,MAS)

# compute density contrast: delta = rho/<rho> - 1
delta /= np.mean(delta, dtype=np.float64);  delta -= 1.0
###################################

######### smooth the field #########
# compute FFT of the filter
W_k = SL.FT_filter(BoxSize, R, grid, Filter, threads)

# smooth the field
delta_smoothed = SL.field_smoothing(delta, W_k, threads)
###################################

########### find mark ##############
# find the value of the smoothed density field in the position of each particle
mark = np.zeros(pos.shape[0], dtype=np.float32)

# find the value of the density field at the positions pos
MASL.CIC_interp(delta, BoxSize, pos, mark)
del delta # we dont need delta anymore; save some memory
###################################

######## compute marked Pk #########
# construct a density field weighting each particle by its overdensity
marked_field = np.zeros((grid,grid,grid), dtype=np.float32)
MASL.MA(pos, marked_field, BoxSize, MAS, W=mark)

# compute marked power spectrum
MPk = PKL.Pk(marked_field, BoxSize, axis, MAS, threads, verbose)

# save 3D marked Pk to file
np.savetxt('My_marked_Pk.txt', np.transpose([MPk.k3D, MPk.Pk[:,0]]))
###################################
```

7.1.5 Velocity power spectrum

Pylians provides a routine, `Pk_theta` that computes the power spectrum of the divergence of a 3D velocity field: $P_{\theta\theta}$, where $\theta = \vec{\nabla} \cdot \vec{V}$. The arguments of the routine are these:

- `Vx`. A 3D numpy float32 array containing the x component of the 3D velocity field, e.g. `Vx = np.zeros((128, 128,128), dtype=np.float32)`.
- `Vy`. A 3D numpy float32 array containing the y component of the 3D velocity field.
- `Vz`. A 3D numpy float32 array containing the z component of the 3D velocity field.
- `BoxSize`. The size of the simulation box. Units here will determine units of output. - `axis`. Axis along which compute the quadrupole, hexadecapole for the theta Pk. If the velocities are in real-space set `axis=0`. If the velocities are in redshift-space set `axis=0`, `axis=1` or `axis=2` if the redshift-space distortions have been placed along the x-axis, y-axis or z-axis, respectively.

- `MAS`. Mass-assignment scheme used to generate the velocity field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the velocity field has not been generated with any of these set it to `'None'`. This is used to correct for the MAS when computing the power spectrum.
- `threads`. Number of openmp threads to be used.

An example of how to use this routine is this:

```
import numpy as np
import Pk_library as PKL

# parameters
BoxSize = 1000.0 #Mpc/h
axis    = 0      #velocity fields in real-space; this variable is not relevant in real-
↪space
MAS     = 'CIC'  #mass-assignment scheme used to create the velocity field
threads = 20     #number of openmp threads to be used

# compute the theta auto-power spectrum
k, Pk, Nmodes = PKL.Pk_theta(Vx,Vy,Vz,BoxSize,axis,MAS,threads)

# k will be in h/Mpc units. Pk will have (km/s)^2*(Mpc/h)^3 considering that the␣
↪velocity field is in km/s
```

Warning: One of the well known problems of computing the velocity power spectrum is empty voxels. When constructing the velocity field, it may happen that no particles reside within (or around) a given voxel. In this case, the velocity field is not well defined. In general, a zero velocity is assigned to that voxel, but that could be a very wrong assumption: for instance, inside voids the the number of particle/galaxy tracers may be low, but the underlying velocity field may be very different to 0. Thus, when using this routine, it is important to make convergence tests (e.g. using different grid sizes for the velocity field) to study the extent and/or presence of this problem.

7.1.6 Momentum power spectrum

Differently to the velocity field, the momentum field, $\vec{p} = \rho\vec{V}$, is well-defined everywhere (even in voxels where there are no particles). Pylians provides the routine `XPk_dv` that computes $P_{\delta\delta}$, $P_{\tilde{\theta}\tilde{\theta}}$, and $P_{\delta\tilde{\theta}}$, where $\delta = \rho/\bar{\rho} - 1$ and $\tilde{\theta} = \vec{\nabla} \cdot (1+\delta)\vec{V}$. The arguments of the function are these:

- `delta`. A 3D numpy float32 array containing the value of the density constrast in each voxel.
- `Vx`. A 3D numpy float32 array containing the x component of the 3D velocity field, e.g. `Vx = np.zeros((128, 128,128), dtype=np.float32)`.
- `Vy`. A 3D numpy float32 array containing the y component of the 3D velocity field.
- `Vz`. A 3D numpy float32 array containing the z component of the 3D velocity field.
- `BoxSize`. The size of the simulation box. Units here will determine units of output. - `axis`. Axis along which compute the quadrupole, hexadecapole for the theta Pk. If the velocities are in real-space set `axis=0`. If the velocities are in redshift-space set `axis=0`, `axis=1` or `axis=2` if the redshift-space distortions have been placed along the x-axis, y-axis or z-axis, respectively.
- `MAS`. Mass-assignment scheme used to generate the velocity field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the velocity field has not been generated with any of these set it to `'None'`. This is used to correct for the MAS when computing the power spectrum.

- `threads`. Number of openmp threads to be used.

An example on how to use this routine is this:

```
import numpy as np
import Pk_library as PKL

# parameters
BoxSize  = 1000.0  #Mpc/h
axis     = 0       #no RSD
MAS      = 'CIC'   #it assumes the density constrast and velocities have been generated␣
↪with the same MAS
threads  = 2       #number of openmp threads

# compute the density constrast and momentum auto- and cross-power spectra
# k will have units of h/Mpc
# Pk_dd will contain the standard power spectrum in (Mpc/h)^3
# Pk_tt will be the momentum auto-power spectrum, defined as above, and with units of␣
↪(km/s)^2*(Mpc/h)^3 in units of velocity field are (km/s)
# Pk_dt will be the density-momentum cross-power spectrum with (km/s)*(Mpc/h)^3 units if␣
↪velocity field has (km/s) units
k, Pk_dd, Pk_tt, Pk_dt, Nmodes = PKL.XPk_dv(delta, Vx, Vy, Vz, BoxSize, axis, MAS,␣
↪threads)
```

7.1.7 Binned power spectrum

Sometimes we may want to compare the power spectrum measured in a simulation versus the theoretical one (e.g. the linear power spectrum). On large scales, the number of modes will be small, so the binning used to compute the power spectrum becomes important when comparing simulations versus theory. Pylians provides the routine `expected_Pk` that will take a power spectrum and will bin it in the same way as is done with the simulations, so a comparison k by k is appropiate.

The ingredients needed are:

- `k_in`. This is an array with the values of k.
- `Pk_in`. This is an array with the values of the power spectrum at `k_in`.
- `BoxSize`. Size of the simulation. If you want to bin the input power spectrum in the same way as the power spectrum measured from a simulation with 1000 Mpc/h, then set `BoxSize = 1000.0`. This parameter determines the fundamental frequency.
- `grid`. The routine will bin the power spectrum according to a mesh with grid x grid x grid voxels. This parameters determines the Nyquist frequency.
- `bins`. The routine will read the input Pk and interpolate it to the k-values sampled in the regular grid. It is desirable to first interpolate the input Pk into a finer 1D mesh to avoid larger errors in the interpolation. This parameter sets the number of bins for that. The more the better, but something around 1000-5000 should be enough.

An example is this:

```
import numpy as np
import Pk_library as PKL

# value of the parameters
```

(continues on next page)

(continued from previous page)

```
f_in    = 'my_linear_Pk.txt'  #input power spectrum
BoxSize = 1000.0 #Mpc/h       #same of box to compute the binned Pk
grid    = 256                 #compute binned Pk using a mesh with grid^3 voxels
bins    = 2000                #number of bins to interpolate the input Pk

# read input power spectrum
k_in, Pk_in = np.loadtxt(f_in, unpack=True)

# get binned Pk: returns k, power spectrum and number of modes in each k-bin
k, Pk, Nmodes = PKL.expected_Pk(k_in, Pk_in, BoxSize, grid, bins)
```

7.2 2D

The routines Pylians provide to compute power spectra for 2 dimensional (images/planes) are these:

7.2.1 Auto-power spectrum

Pylians can also compute auto-power spectra of images/planes through the `Pk_plane` routine. The ingredients needed are:

- `delta`. This should be a 2D numpy float32 array, like `delta = np.zeros((128,128), dtype=np.float32)`.
- `BoxSize`. The size of the plane.
- `MAS`. Mass-assignment scheme used to generate the 2D density field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the density field has not been generated with any of these, set it to `'None'`. This is used to correct for the MAS when computing the power spectrum.
- `threads`. Number of openmp threads to be used in the calculation.

An example of how to utilize this function is this:

```
import numpy as np
import Pk_library as PKL

# parameters
grid    = 128     #the map will have grid^2 pixels
BoxSize = 1000.0  #Mpc/h
MAS     = 'None'  #MAS used to create the image; 'NGP', 'CIC', 'TSC', 'PCS' o 'None'
threads = 1       #number of openmp threads

# create an empty image
delta = np.zeros((grid,grid), dtype=np.float32)

# compute the Pk of that image
Pk2D = PKL.Pk_plane(delta, BoxSize, MAS, threads)

# get the attributes of the routine
k      = Pk2D.k      #k in h/Mpc
Pk     = Pk2D.Pk     #Pk in (Mpc/h)^2
Nmodes = Pk2D.Nmodes #Number of modes in the different k bins
```

7.2.2 Cross-power spectrum

Pylians provide the routine `XPk_plane` to compute cross-power spectrum between two images. The ingredients needed are:

delta1, delta2, BoxSize, MAS1=None, MAS2=None, threads=1):

- `delta1`. A 2D numpy float32 array containing the data of the first image.
- `delta2`. A 2D numpy float32 array containing the data of the second image.
- `BoxSize`. Size of the plane. Note that the size of both images should be the same.
- `MAS1`. The MAS (mass assignment scheme) employed to construct the first image, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the density field has not been generated with any of these, set it to `'None'`. This is used to correct for the MAS when computing the power spectrum.
- `MAS2`. Same as `MAS1` but for the second image.
- `threads`. Number of openmp threads to use.

An example of how to utilize this routine is this:

```
import numpy as np
import Pk_library as PKL

# parameters
BoxSize = 1000.0 #Mpc/h
MAS1    = 'CIC'
MAS2    = 'None'
threads = 1

# compute cross-power spectrum between two images
XPk2D = PKL.XPk_plane(delta1, delta2, BoxSize, MAS1, MAS2, threads)

# get the attributes of the routine
k      = XPk2D.k        #k in h/Mpc
Pk     = XPk2D.Pk       #auto-Pk of the two maps in (Mpc/h)^2
Pk1    = Pk[:,0]        #auto-Pk of the first map in (Mpc/h^2)
Pk2    = Pk[:,1]        #auto-Pk of the second map in (Mpc/h^2)
XPk    = XPk2D.XPk      #cross-Pk in (Mpc/h)^2
r      = XPk2D.r        #cross-correlation coefficient
Nmodes = XPk2D.Nmodes   #number of modes in each k-bin
```

CHAPTER

EIGHT

CORRELATION FUNCTION

Pylians provides several routines to compute auto- and cross-correlation functions.

Note: For the time being, Pylians routines to compute correlation functions only support 3D fields.

8.1 Auto-correlation function

Pylians can be used to efficiently compute correlation functions of a generic field (e.g. total matter, CDM, gas, halos, neutrinos, CDM+gas, galaxies...etc). The ingredients needed are:

- `delta`. This is the density contrast field. It should be a 3 dimensional float numpy array such `delta = np.zeros((grid, grid, grid), dtype=np.float32)`. See *Density fields* on how to compute density fields using Pylians.
- `BoxSize`. Size of the periodic box. The units of the output radii will depend on this.
- `MAS`. Mass-assignment scheme used to generate the density field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the density field has not been generated with any of these set it to `'None'`.
- `axis`. Axis along which compute the quadrupole, hexadecapole. If the field is in real-space set `axis=0`. If the field is in redshift-space set `axis=0`, `axis=1` or `axis=2` if the redshift-space distortions have been placed along the x-axis, y-axis or z-axis, respectively.
- `threads`. This routine is openmp parallelized. Set this to the maximum number of cores per node available in your machine.

An example on how to use the routine is this:

```
import numpy as np
import Pk_library as PKL

# correlation function parameters
BoxSize = 1000.0 #Mpc/h
MAS     = 'CIC'
axis    = 0
threads = 16

# compute the correlation function
CF      = PKL.Xi(delta, BoxSize, MAS, axis, threads)

# get the attributes
```

(continues on next page)

(continued from previous page)

```
r      = CF.r3D      #radii in Mpc/h
xi0    = CF.xi[:,0]  #correlation function (monopole)
xi2    = CF.xi[:,1]  #correlation function (quadrupole)
xi4    = CF.xi[:,2]  #correlation function (hexadecapole)
Nmodes = CF.Nmodes3D #number of modes
```

Note: This routine uses a FFT approach that allows a very computationally efficient calculation of the correlation function. However, if the number density of the tracers is very low (i.e. the density field is very sparse) this function may produce strange results. In this case it is better to use the traditional Landy-Szalay routine also available in Pylians.

8.2 Cross-correlation function

The routine `XXi` can be used to compute the cross-correlation function between two generic fields. The ingredients needed are:

- `delta1`. A 3D numpy float32 array containing the value of the first density field; e.g. `delta1 = np.zeros((128,128,128), dtype=np.float32)`.
- `delta2`. A 3D numpy float32 array containing the value of the second density field.
- `BoxSize`. Size of the periodic box. The units of the output radii will depend on this. Note that both fields have to have the same size.
- `MAS`. Mass-assignment scheme used to generate the 3D density field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the density field has not been generated with any of these set it to `'None'`. In this case, this variable should be a tuple with 2 values, once for each field.
- `axis`. Axis along which compute the quadrupole, hexadecapole. If the field is in real-space set `axis=0`. If the field is in redshift-space set `axis=0`, `axis=1` or `axis=2` if the redshift-space distortions have been placed along the x-axis, y-axis or z-axis, respectively.
- `threads`. Number of openmp threads to be used.

An example of the usage of this routine is this:

```
import numpy as np
import Pk_library as PKL

# correlation function parameters
BoxSize = 1000.0 #Mpc/h
MAS     = ['CIC', 'None']
axis    = 0
threads = 16

# compute cross-correlaton function of the two fields
CCF = PKL.XXi(delta1, delta2, BoxSize, MAS, axis, threads)

# get the attributes
r      = CCF.r3D      #radii in Mpc/h
xxi0   = CCF.xi[:,0]  #monopole
xxi2   = CCF.xi[:,1]  #quadrupole
```

(continues on next page)

(continued from previous page)

```
xxi4   = CCF.xi[:,2]  #hexadecapole
Nmodes = CCF.Nmodes3D #number of modes
```

Note: This routine uses a FFT approach that allows a very computationally efficient calculation of the cross-correlation function. However, if the number density of the tracers is very low (i.e. the density field is very sparse) this function may produce strange results. In this case it is better to use the traditional Landy-Szalay routine also available in Pylians.

CHAPTER

NINE

BISPECTRUM

Pylians can compute bispectra of 3D density fields (total matter, CDM, gas, halos, galaxies...etc). The ingredients needed are:

- `delta`. This is the density contrast field. It should be a 3 dimensional float numpy array such `delta = np.zeros((grid, grid, grid), dtype=np.float32)`. See *Density fields* on how to compute density fields using Pylians.
- `BoxSize`. Size of the periodic box. The units of the output bispectrum depend on this.
- `k1`. The wavenumber of the first side of the considered triangle. Use units in correspondence with `BoxSize`.
- `k2`. The wavenumber of the second size of the considered triangle. Use units in correspondence with `BoxSize`.
- `theta`. This is a numpy array containing the different angles between `k1` and `k2`.
- `MAS`. Mass-assignment scheme used to generate the density field, if any. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`. If the density field has not been generated with any of these set it to `'None'`.
- `threads`. The bispectrum code is openmp parallelized. Set this to the maximum number of cpus per node.

Pylians computes the amplitude of the bispectrum, and reduced bispectrum, for triangles that have sides `k1` and `k2` and different considered angles between them.

```
import numpy as np
import Pk_library as PKL

BoxSize = 1000.0 #Size of the density field in Mpc/h
k1      = 0.5    #h/Mpc
k2      = 0.6    #h/Mpc
MAS     = 'CIC'
threads = 1
theta   = np.linspace(0, np.pi, 25) #array with the angles between k1 and k2

# compute bispectrum
BBk = PKL.Bk(delta, BoxSize, k1, k2, theta, MAS, threads)
Bk  = BBk.B     #bispectrum
Qk  = BBk.Q     #reduced bispectrum
k   = BBk.k     #k-bins for power spectrum
Pk  = BBk.Pk    #power spectrum
```

CHAPTER

TEN

VOIDS

Pylians can be used to identify voids in a generic density field (e.g. total matter, CDM, gas, halos, neutrinos, CDM+gas, galaxies...etc). The ingredients needed are:

- `delta`. This is the considered field; usually the density constrast: $\delta = \rho/\bar{\rho} - 1$. It should be a 3 dimensional float numpy array such `delta = np.zeros((grid, grid, grid), dtype=np.float32)`. See *Density fields* on how to generate density fields using Pylians.
- `BoxSize`. Size of the periodic box. The units of the output power spectrum depend on this.
- `threshold`. The routine will identify voids with mean overdensity `(1+threshold)`. This value is typically -0.7 or -0.8, but can be higher (e.g. -0.5), depending on your needs.
- `Radii`. This is a `np.float32` 1-dimension array containing the radii of the voids to identify. It doesn't need to be sorted. In general, the minimum void size should be ~4x-5x the grid size. E.g. if you have box of 1000 Mpc/h and a grid with `1000^3` cells, the minimum void size should be of 4-5 Mpc/h. It is better to choose the Radii such as their are multiples of the grid size. For the previous examples this will be good: `Radii = np.array([4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 25, 28, 31, 34, 37, 40, 45, 50, 55], dtype=np.float32)`.
- `threads1`. The void finder routine is openmp parallelized. Set this to the maximum number of cpus per node.
- `threads2`. Some routines are slower using all available cores. for those, we use a smaller number of cores. This number is typically 4 at most.
- `void_field`. The routine can return a 3-dimension field filled with 0 (no void) and 1 (void) from the identified voids. If you want this set `void_field=True`.

The void finder routine works as follows:

```
import numpy as np
import void_library as VL

# parameters of the void finder
BoxSize    = 1000.0 #Mpc/h
threshold  = -0.7
Radii      = np.array([5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31,
                       33, 35, 37, 39, 41, 44, 47, 50, 53, 56], dtype=np.float32) #Mpc/h
threads1   = 16
threads2   = 4
void_field = False

# identify voids
V = VL.void_finder(field, BoxSize, threshold, Radii, threads1, threads2, void_
↪field)
```

(continues on next page)

(continued from previous page)

```
void_pos    = V.void_pos    #positions of the void centers
void_radius = V.void_radius #radius of the voids
VSF_R       = V.Rbins       #bins in radius for VSF(void size function)
VSF         = V.void_vsf    #VSF (#voids/volume/dR)
if void_field:  void_field  = V.void_field
```

10.1 Method

Pylians3 uses the spherical overdensity void finder described in Banerjee & Dalal 2016. We provide an example on how spherical voids can be easily identified with the void finder routine available in Pylians3:

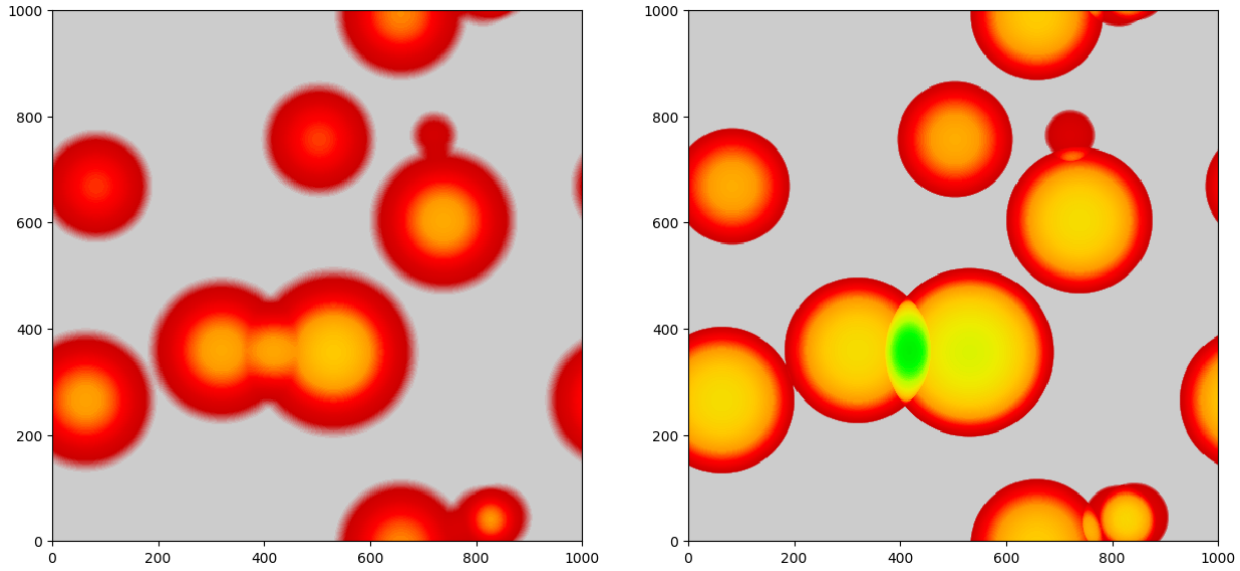
```
# This script takes an uniform density field and places random spheres
# of random sizes with a profile of delta(r)=-1*(1-(r/R)^3).
# Then it identifies the voids using the void finder. Finally, it plots the
# average density field across the entire box of the input and recovered void field.
import numpy as np
import sys,os,time
import void_library as VL
from pylab import *
from matplotlib.colors import LogNorm

################################ INPUT ##########################################
BoxSize = 1000.0 #Mpc/h
Nvoids  = 10     #number of random voids
dims    = 512    #grid resolution to find voids

threshold = -0.5 #for delta(r)=-1*(1-(r/R)^3)

Rmax = 200.0   #maximum radius of the input voids
Rmin = 20.0    #minimum radius of the input voids
bins = 50      #number of radii between Rmin and Rmax to find voids

threads1 = 16 #openmp threads
threads2 = 4

f_out = 'Spheres_test.png'
#################################################################################


# create density field with random spheres
V = VL.random_spheres(BoxSize, Rmin, Rmax, Nvoids, dims)
delta = V.delta

# find voids
Radii = np.logspace(np.log10(Rmin), np.log10(Rmax), bins+1, dtype=np.float32)
V2 = VL.void_finder(delta, BoxSize, threshold, Radii,
                    threads1, threads2, void_field=True)
delta2 = V2.in_void
```

(continues on next page)

(continued from previous page)

```
# print the positions and radius of the generated voids
pos1 = V.void_pos
R1   = V.void_radius
pos2 = V2.void_pos
R2   = V2.void_radius

print('          X       Y       Z       R')
for i in range(Nvoids):
    dx = pos1[i,0]-pos2[:,0]
    dx[np.where(dx>BoxSize/2.0)] -= BoxSize
    dx[np.where(dx<-BoxSize/2.0)]+= BoxSize

    dy = pos1[i,1]-pos2[:,1]
    dy[np.where(dy>BoxSize/2.0)] -= BoxSize
    dy[np.where(dy<-BoxSize/2.0)]+= BoxSize

    dz = pos1[i,2]-pos2[:,2]
    dz[np.where(dz>BoxSize/2.0)] -= BoxSize
    dz[np.where(dz<-BoxSize/2.0)]+= BoxSize

    d = np.sqrt(dx*dx + dy*dy + dz*dz)
    index = np.where(d==np.min(d))[0]
    pos, R = pos2[index][0], R2[index][0]

    print('\nVoid %02d'%i)
    print("Actual:     %6.2f  %6.2f  %6.2f  %6.2f"\
            %(pos1[i,0], pos1[i,1], pos1[i,2], R1[i]))
    print("Identified: %6.2f  %6.2f  %6.2f  %6.2f"\
            %(pos[0],    pos[1],    pos[2],   R))


############# plot results #############
fig = figure(figsize=(15,7))
ax1,ax2 = fig.add_subplot(121), fig.add_subplot(122)

# plot the density field of the random spheres
ax1.imshow(np.mean(delta[:,:,:],axis=0),
           cmap=get_cmap('nipy_spectral'),origin='lower',
           vmin=-1, vmax=0.0, extent=[0, BoxSize, 0, BoxSize])

# plot  the void field identified by the void finder
ax2.imshow(np.mean(delta2[:,:,:],axis=0),
           cmap=get_cmap('nipy_spectral_r'),origin='lower',
           vmin=0, vmax=1.0, extent=[0, BoxSize, 0, BoxSize])

savefig(f_out, bbox_inches='tight')
show()
########################################
```

The above script generates random spheres with density profiles given by $\delta(r) = -\left[1 - (\frac{r}{R})^3\right]$ in a given cosmological volume. Note that for this density profile, the average overdensity at the void radius, R, is -0.5. The script then identifies the voids in that density field and finally plot the results. A figure like this should be obtained:



The left panel shows the projected density field of the generated random uniform spheres. The right panel displays the projected field of the identified voids. Note that, visually, density profiles look different in the two cases because the void finder set to 1 every voxel that belongs to a void, while the random spheres follow the above density profiles. The code also outputs the positions and radii of the generated and identified spheres.

CHAPTER

ELEVEN

REDSHIFT-SPACE DISTORTIONS

Pylians provides the routine `pos_redshift_space` to displace particle positions from real-space to redshift-space. The arguments of that function are:

- `pos`. This is an array with the co-moving positions of the particles. Should be float numpy array. Notice that this array will be overwritten with the positions of the particles in redshift-space. So if you want to keep the positions of the original particles, is better to pass a copy of this array: e.g. `pos_RSD = np.copy(pos)`. Units should be Mpc/h.
- `vel`. This is an array with the peculiar velocities of the particles. Should be a float numpy array. Units should be km/s
- `BoxSize`. Size of the simulation box. Units should be Mpc/h
- `Hubble`. Value of the Hubble constant at redshift `redshift`. Units should be (km/s)/(Mpc/h).
- `redshift`. The considered redshift.
- `axis`. Redshift-space distortions are going to be place along the x-(axis=0), y-(axis=1) or z-(axis=2) axis.

```
import redshift_space_library as RSL

# move particles to redshift-space. After this call, pos will contain the
# positions of the particles in redshift-space
RSL.pos_redshift_space(pos, vel, BoxSize, Hubble, redshift, axis)
```

CHAPTER

TWELVE

PLOTS

Pylians provides a set of routines to quickly make plots of density fields of simulations. For instance, to generate the density field of a slice of a Gadget N-body snapshot, the ingredients needed are:

- `snapshot`. This is the name of the snapshot. For instance, `snapdir_004/snap_004`. Note that only the prefix is needed. If e.g. `snapdir_004` is a folder that contains multiple subfiles, `snapdir_004/snap_004.0`, `snapdir_004/snap_004.1`...etc, `snapshot` should be set to `snapdir_004/snap_004`.
- `x_min`, `x_max`, `y_min`, `y_max`, `z_min`, `z_max`. These are the coordinates of the considered region.
- `grid`. Resolution of the image, that will have grid x grid pixels.
- `ptypes`. Particle type over which compute the density field. It can be individual types, `[0]` (gas), `[1]` (cold dark matter), `[2]` (neutrinos), `[3]` (particle type 3), `[4]` (stars), `[5]` (black holes), or combinations. E.g. `[0,1]` (gas+cold dark matter), `[0,4]` (gas+stars), `[0,1,2,4]` (gas+CDM+neutrinos+stars). For all components (total matter) use `[0,1,2,3,4,5]` or `[-1]`.
- `plane`. Plane over which project the region. Can be `'XY'`, `'XZ'` or `'YZ'`.
- `MAS`. Mass-assignment scheme used to generate the density field. Possible options are `'NGP'`, `'CIC'`, `'TSC'`, `'PCS'`.
- `save_df`. Whether save the density field or not. It can be useful to save the field, and don't have to recompute it, when only changing the colors, overdensities values...etc.

One example on how to use Pylians to plots the density field of a snapshot is this

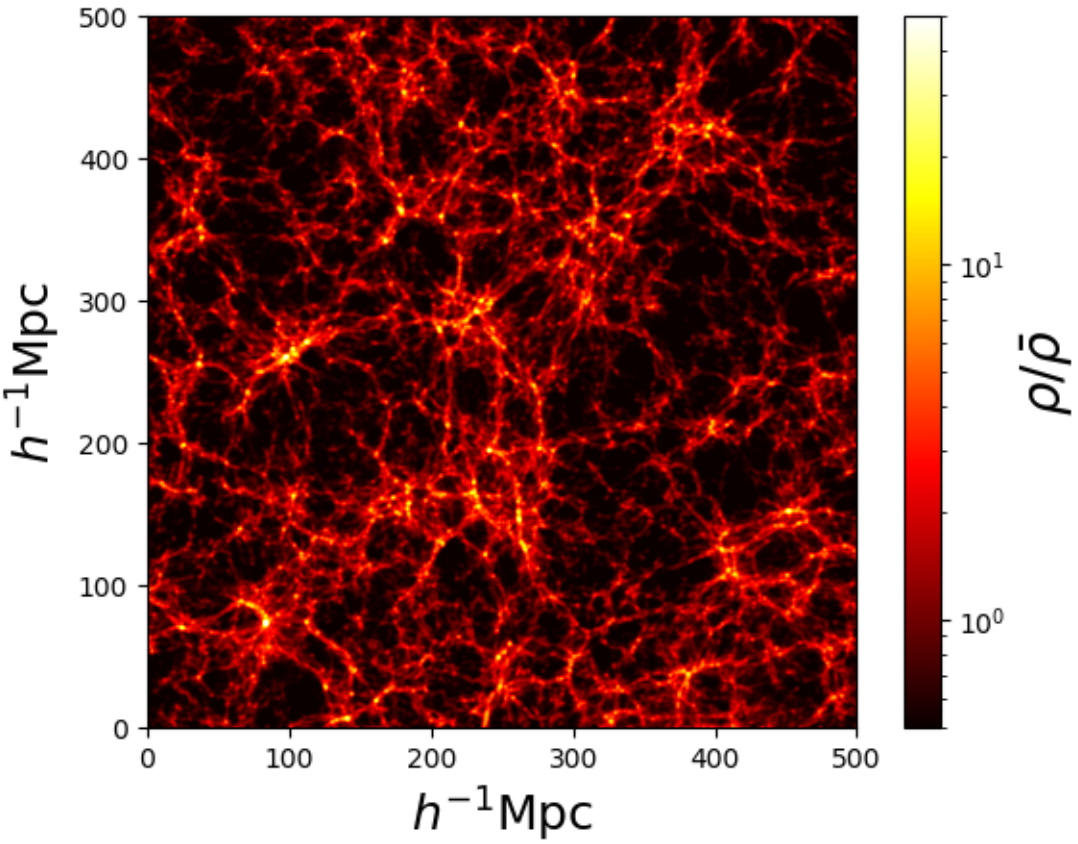
```
import numpy as np
import plotting_library as PL
from pylab import *
from matplotlib.colors import LogNorm

#snapshot name
snapshot = '/mnt/ceph/users/fvillaescusa/Quijote/Snapshots/latin_hypercube_HR/0/snapdir_
↪004/snap_004'

# density field parameters
x_min, x_max = 0.0, 500.0
y_min, y_max = 0.0, 500.0
z_min, z_max = 0.0, 20.0
grid         = 1024
ptypes       = [1]   # 0-Gas, 1-CDM, 2-NU, 4-Stars; can deal with several species
plane        = 'XY'  #'XY','YZ' or 'XZ'
MAS          = 'PCS' #'NGP', 'CIC', 'TSC', 'PCS'
save_df      = True  #whether save the density field into a file
```

(continues on next page)

(continued from previous page)

```
# image parameters
fout            = 'Image.png'
min_overdensity = 0.5      #minimum overdensity to plot
max_overdensity = 50.0     #maximum overdensity to plot
scale           = 'log' #'linear' or 'log'
cmap            = 'hot'


# compute 2D overdensity field
dx, x, dy, y, overdensity = PL.density_field_2D(snapshot, x_min, x_max, y_min, y_max,
                                                z_min, z_max, grid, ptypes, plane, MAS,␣
↪save_df)


# plot density field
print('\nCreating the figure...')
fig = figure()    #create the figure
ax1 = fig.add_subplot(111)

ax1.set_xlim([x, x+dx])  #set the range for the x-axis
ax1.set_ylim([y, y+dy])  #set the range for the y-axis

ax1.set_xlabel(r'$h^{-1}{\rm Mpc}$',fontsize=18)  #x-axis label
ax1.set_ylabel(r'$h^{-1}{\rm Mpc}$',fontsize=18)  #y-axis label

if min_overdensity==None:  min_overdensity = np.min(overdensity)
if max_overdensity==None:  max_overdensity = np.max(overdensity)

overdensity[np.where(overdensity<min_overdensity)] = min_overdensity

if scale=='linear':
      cax = ax1.imshow(overdensity,cmap=get_cmap(cmap),origin='lower',
                       extent=[x, x+dx, y, y+dy], interpolation='bicubic',
                       vmin=min_overdensity,vmax=max_overdensity)
else:
      cax = ax1.imshow(overdensity,cmap=get_cmap(cmap),origin='lower',
                       extent=[x, x+dx, y, y+dy], interpolation='bicubic',
                       norm = LogNorm(vmin=min_overdensity,vmax=max_overdensity))

cbar = fig.colorbar(cax)
cbar.set_label(r"$\rho/\bar{\rho}$",fontsize=20)
savefig(fout, bbox_inches='tight')
close(fig)
```

The above script, on one of the Quijote simulations produces the following image:


500
400
300
200
100
0
0
100
200
300
400
500
h^{-1}Mpc
h^{-1}Mpc
10^1
10^0
$\rho/\bar{\rho}$

CHAPTER

THIRTEEN

COSMOLOGY

Pylians provide a set of routines to carry out simple cosmological calculations.

13.1 Comoving distance

The comoving distance to redshift z can be computed as:

```
import cosmology_library as CL

z       = 1.0
Omega_m = 0.3175
Omega_l = 0.6825

# compute the comoving distance to redshift z in Mpc/h
r = CL.comoving_distance(z, Omega_m, Omega_l)  #Mpc/h
```

13.2 Linear growth factor

The linear growth factor to redshift z can be computed as:

```
import cosmology_library as CL

z       = 1.0
Omega_m = 0.3175
Omega_l = 0.6825

# compute the linear growth factor
D = CL.linear_growth_factor(z, Omega_m, Omega_l)
```

13.3 Halofit

From a linear power spectrum at z=0, Pylians can find the non-linear matter power spectrum halofit by Takahashi 2012 as

```
import numpy as np
import cosmology_library as CL

z       = 1.0
Omega_m = 0.3175
Omega_l = 0.6825

# read the linear power spectrum
k_lin, Pk_lin = np.loadtxt('my_Pk_file_z=0.txt', unpack=True)

# find the non-linear power spectrum from halofit
Pk_nl = CL.Halofit_12(Omega_m, Omega_l, z, k_lin, Pk_lin)
```

CHAPTER

FOURTEEN

HALO MASS FUNCTION

Pylians provides the routine `MF_theory` to compute the halo mass function of a given model. The arguments of this function are:

k_in, Pk_in, OmegaM, Masses, author, bins=10000, z=0, delta=200.0

- `k`. 1D numpy array with the value of the linear matter power spectrum wavenumbers.
- `Pk`. 1D numpy array with the amplitude of the linear matter power spectrum on the wavenumbers `k`.
- `OmegaM`. Value of $\Omega_{\rm m}$.
- `Masses`. 1D numpy array with the value of the halo masses over which compute the halo mass function.
- `author`. The *model* for the halo mass function. Options are: `ST`, `Tinker`, `Tinker10`, `Crocce`, `Jenkins`, `Warren`, `Watson`, `Watson_FoF`, `Angulo`.
- `bins`. In order to carry out the integrals, the k bins need to be sorted and equally spaced in log10. This parameter determines the number of bins to use. The more the better, but a very large number will have very little impact. Default 10000.
- `z`. Redshift at which to estimate the halo mass function. Only needed for the `Tinker`, `Tinker10`, and `Crocce` mass functions.
- `delta`. The overdensity value. Default is 200. Only needed for `Tinker` and `Tinker10`.

Note: For cosmologies with massive neutrinos, $\Omega_{\rm m}$ should be set to $\Omega_{\rm c}+\Omega_{\rm b}$ and the linear power spectrum should be the CDM+baryons linear power spectrum; see e.g. 1311.1212 and 1311.1514.

An example of how to use this routine is this:

```
import numpy as np
import mass_function_library as MFL

# halo mass function parameters
f_Pk   = 'Pk_linear_z=0.txt'  #file with linear Pk
OmegaM = 0.3175
Masses = np.logspace(11, 15, 100) #array with halo masses
author = 'ST'   #Sheth-Tormen halo mass function
bins   = 10000  #number of bins to use for Pk
z      = 0.0    #redshift; only used for Tinker, Tinker10 and Crocce
delta  = 200.0  #overdensity; only for Tinker and Tinker10

# read linear matter Pk
k, Pk = np.loadtxt(f_Pk, unpack=True)
```

(continues on next page)

(continued from previous page)

```
# compute halo mass function
HMF = MFL.MF_theory(k, Pk, OmegaM, Masses, author, bins, z, delta)
```

14.1 variance

Pylians provides the routine `sigma` that can be used to compute σ_R, defined as

$$\sigma_R = \int_0^\infty P(k) W(k,R)^2 k^2/(2\pi^2)$$

where $W(k,R)$ is the Fourier transform of a top-hat function with radius R:

$$W(k,R) = \frac{3[\sin(kR) - kR\cos(kR)]}{(kR)^3}$$

The most standard applicaiton of this routine is to compute the value of σ_8 given a linear power spectrum:

```
import numpy as np
import mass_function_library as MFL

# read linear power spectrum
k, Pk = np.loadtxt('My_linear_Pk.txt', unpack=True)

# compute the value of sigma_8
sigma_8 = MFL.sigma(k, Pk, 8.0)
```

CHAPTER

FIFTEEN

GAUSSIAN DENSITY FIELDS

Pylians provide a few routines to generate Gaussian density fields either in 2D or 3D. The ingredients needed are:

- `grid`. The generated Gaussian density field will have grid x grid pixels in 2D or grid x grid x grid voxels in 3D.
- `k` . 1D float32 numpy array containing the k-values of the input power spectrum.
- `Pk`. 1D float32 numpy array containing the Pk-values of the input power spectrum.
- `Rayleigh_sampling`. Where Rayleigh sampling the modes amplitudes when generating the Gaussian field. If `Rayleigh_sampling=0` the Gaussian field will not have cosmic variance. Set `Rayleigh_sampling=1` for standard Gaussian density fields.
- `seed`. Integer for the random seed of the map.
- `BoxSize`. Size of the region over which to generate the field. Units should be compatible with those of `Pk`.
- `threads`. Number of openmp threads. Only used when FFT the field from Fourier space to configuration space.
- `verbose`. Whether output some information.

An example on how to generate these fields in 2D and 3D is this:

```
import numpy as np
import density_field_library as DFL

grid             = 128    #grid size
BoxSize          = 1000.0 #Mpc/h
seed             = 1      #value of the initial random seed
Rayleigh_sampling = 0     #whether sampling the Rayleigh distribution for modes␣
↪amplitudes
threads          = 1      #number of openmp threads
verbose          = True   #whether to print some information

# read power spectrum; k and Pk have to be floats, not doubles
k, Pk = np.loadtxt('my_Pk.txt', unpack=True)
k, Pk = k.astype(np.float32), Pk.astype(np.float32)

# generate a 2D Gaussian density field
df_2D = DFL.gaussian_field_2D(grid, k, Pk, Rayleigh_sampling, seed,
                              BoxSize, threads, verbose)

# generate a 3D Gaussian density field
df_3D = DFL.gaussian_field_3D(grid, k, Pk, Rayleigh_sampling, seed,
                              BoxSize, threads, verbose)
```

CHAPTER

SIXTEEN

INTEGRALS

Pylians provide routines to carry out numerical integrals in a more efficient way than scipy integrate. The philosophy is that to compute the integral $\int_a^b f(x)dx$ the user passes the integrator two arrays, one with some values of x between a and b and another with the values of $f(x)$ at those positions. The integrator will interpolate internally the input data to evaluate the function at an arbitrary position x. Pylians implements in c the fortran odeint function and wrap in python through cython.

For instance, to compute $\int_0^5 (3x^3 + 2x + 5)dx$ one would do

```
import numpy as np
import integration_library as IL

# integral value, its limits and precision parameters
yinit = np.zeros(1, dtype=np.float64)
x1    = 0.0
x2    = 5.0
eps   = 1e-8
h1    = 1e-10
hmin  = 0.0

# integral method and integrand function
function = 'linear'
bins     = 1000
x        = np.linspace(x1, x2, bins)
y        = 3*x**2 + 2*x + 5

I = IL.odeint(yinit, x1, x2, eps, h1, hmin, x, y, function, verbose=True)[0]
```

The value of integral is stored in `I`. The `odeint` routine needs the following ingredients:

- `yinit`. The value of the integral is stored in this variable. Should be a 1D double numpy array with one single element equal to 0. If several integrals are being computed sequentially this variable need to be declared for each integral.
- `x1`. Lower limit of the integral.
- `x2`. Upper limit of the integral.
- `eps`. Maximum local relative error tolerated. Typically set its value to be 1e8-1e10 lower than the value of the integral. Verify the convergence of the results by studying the dependence of the integral on this number.
- `h1`. Initial guess for the first time step (cannot be 0).
- `hmin`. Minimum allowerd time step (can be 0).
- `verbose`. Set it to `True` to print some information on the integral computation.

There are two main methods to carry out the integral, depending on how the interpolation is performed.

- **`function = 'linear'`. The function is evaluated by interpolating linearly the input values of x and $y = f(x)$.**
  - `x`. 1D double numpy array containing the input, equally spaced, values of x.
  - `y`. 1D double numpy array containing the values of $y = f(x)$ at the `x` array positions.
- **`function = 'log'`. The function is evaluated by interpolating logaritmically the input values of x and $y = f(x)$.**
  - `x`. 1D double numpy array containing the input, equally spaced in log, values of $\log_{10}(x)$.
  - `y`. 1D double numpy array containing the values of $y = f(x)$ at the `x` array positions.

An example of using the log-interpolation to compute the integral $\int_1^2 e^x dx$ is this

```
import numpy as np
import integration_library as IL

# integral value, its limits and precision parameters
yinit = np.zeros(1, dtype=np.float64)
x1    = 1.0
x2    = 2.0
eps   = 1e-10
h1    = 1e-12
hmin  = 0.0

# integral method and integrand function
function = 'log'
bins     = 1000
x        = np.logspace(np.log10(x1), np.log10(x2), bins)
y        = np.exp(x)

I = IL.odeint(yinit, x1, x2, eps, h1, hmin, np.log10(x), y,
              function, verbose=True)[0]
```

Warning: Be careful when using the log-interpolation, since the code will crash if a zero or negative value is encounter.

The user can create its own function to avoid evaluating the integrand via interpolations. This function has to be placed in the file library/integration_library/integration_library.pyx (see linear and sigma functions as examples). After that, a new function call has to be created in the function odeint of that file (see linear and log as examples).

CHAPTER

SEVENTEEN

GADGET

Pylians provides a few routines to work with Gadget snapshots.

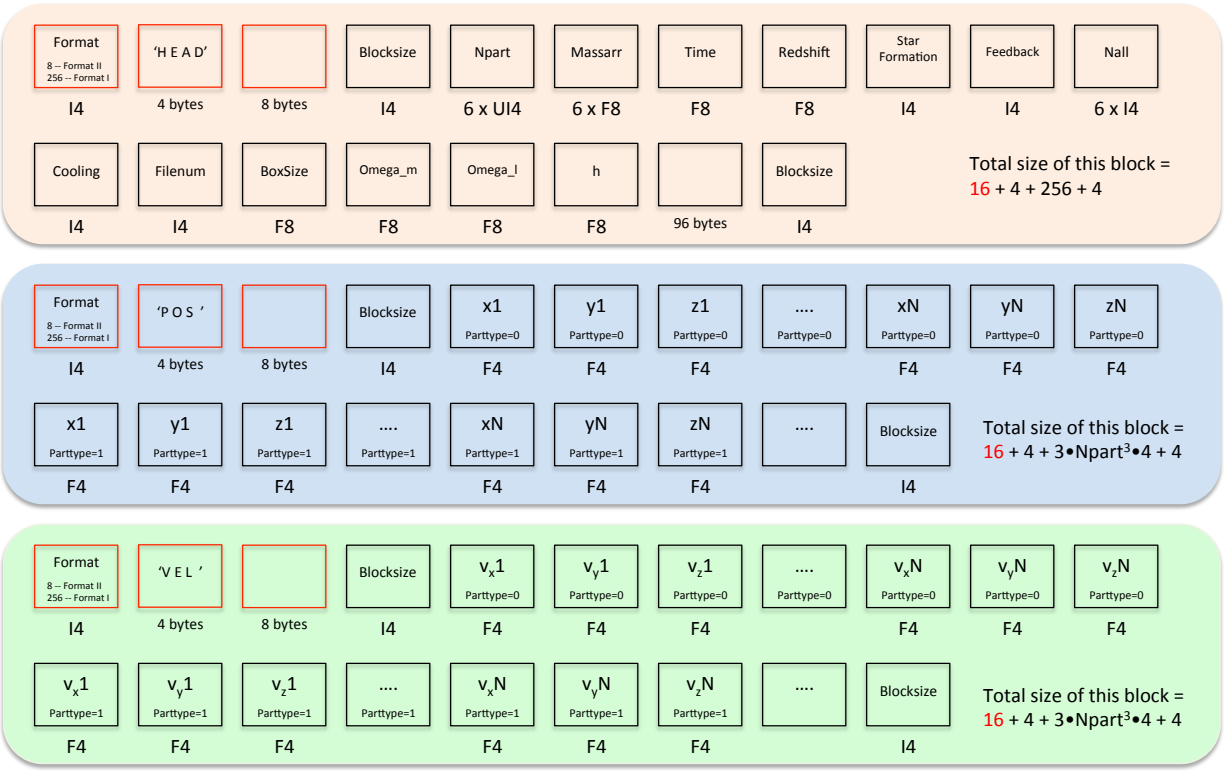
17.1 Snapshots

The routine library `readgadget` can be used to read generic Gadget snapshots, with format I, II or hdf5. An example is this:

```
import readgadget

# input files
snapshot = '/home/fvillaescusa/Quijote/Snapshots/h_p/snapdir_002/snap_002'
ptype    = [1] #[1](CDM), [2](neutrinos) or [1,2](CDM+neutrinos)

# read header
header   = readgadget.header(snapshot)
BoxSize  = header.boxsize/1e3  #Mpc/h
Nall     = header.nall         #Total number of particles
Masses   = header.massarr*1e10 #Masses of the particles in Msun/h
Omega_m  = header.omega_m      #value of Omega_m
Omega_l  = header.omega_l      #value of Omega_l
h        = header.hubble       #value of h
redshift = header.redshift     #redshift of the snapshot
Hubble   = 100.0*np.sqrt(Omega_m*(1.0+redshift)**3+Omega_l)#Value of H(z) in km/s/(Mpc/h)

# read positions, velocities and IDs of the particles
pos = readgadget.read_block(snapshot, "POS ", ptype)/1e3 #positions in Mpc/h
vel = readgadget.read_block(snapshot, "VEL ", ptype)     #peculiar velocities in km/s
ids = readgadget.read_block(snapshot, "ID  ", ptype)-1   #IDs starting from 0
```

Note: While `readgadget` can read generic N-nody outputs from Gadget, it may only be able to read a few blocks from hydrodynamic simulations. In this case is better to modify the library to read the particular fields that may be unique to your simulation.

The scheme below shows the traditional structure of the Format I and Format II Gadget snapshots.



17.2 Halo catalogues

The library `readfof` can be used to read Friends-of-Friends (FoF) halo catalogues that are written in Gadget format. An example is this:

```
import readfof

# input files
snapdir = '/home/fvillaescusa/Quijote/Halos/s8_p/145/' #folder hosting the catalogue
snapnum = 4                                            #redshift 0

# determine the redshift of the catalogue
z_dict = {4:0.0, 3:0.5, 2:1.0, 1:2.0, 0:3.0}
redshift = z_dict[snapnum]

# read the halo catalogue
FoF = readfof.FoF_catalog(snapdir, snapnum, long_ids=False,
                          swap=False, SFR=False, read_IDs=False)

# get the properties of the halos
pos_h = FoF.GroupPos/1e3            #Halo positions in Mpc/h
```

(continues on next page)

(continued from previous page)

```
mass  = FoF.GroupMass*1e10          #Halo masses in Msun/h
vel_h = FoF.GroupVel*(1.0+redshift) #Halo peculiar velocities in km/s
Npart = FoF.GroupLen                #Number of CDM particles in the halo
```

CHAPTER

EIGHTEEN

ICS POWER SPECTRUM

In some cases we may want to compute the power spectrum of the initial density field of a simulation. The particle positions from the initial conditions can be used to estimate that. However, this will only be an approximation.

We have modified the N-GenIC code by Volker Springel to output the modes amplitudes, phases, and wavenumbers of the initial density field; you can find this N-GenIC version here.

When generating initial conditions with that code (switch the `DOUTPUT_DF` flag in the Makefile), several files will be written starting with `Amplitudes_`, `Coordinates_`, and `Phases_`. Pylians provides the routine `Pk_NGenIC_IC_field` that can be used to compute the power spectrum of the initial field from those files. The arguments of that function are these:

- `f_coordinates`. The prefix of the files containing the modes coordinates. Note that only the prefix is needed, not the whole file name (or file names when multiple files).
- `f_amplitudes`. The prefix of the files containing the modes amplitudes. Note that only the prefix is needed, not the whole file name (or file names when multiple files).
- `BoxSize`. Size of the simulation box in Mpc/h.

An example of how to use the routine is this

```
import numpy as np
import Pk_library as PKL

# parameters
f_coordinates = 'Coordinates_ptype_1' #modes coordinates of the dark matter (ptype 1)
f_amplitudes  = 'Amplitudes_ptype_1'  #modes amplitudes of the dark matter (ptype 1)
BoxSize       = 512.0                 #Mpc/h

# compute Pk of the initial field
# k will be in h/Mpc, while P(k) will have (Mpc/h)^3 units
k, Pk, Nmodes = PKL.Pk_NGenIC_IC_field(f_coordinates, f_amplitudes, BoxSize)
```

CHAPTER

NINETEEN

TUTORIALS

We refer the user to this website for a series of tutorials on how to read and manipulate data from numerical simulations that using Pylians.

CHAPTER

TWENTY

LICENSE

CHAPTER

TWENTYONE

CITATION

If you use Pylians consider citing it thourgh:

```
@MISC{Pylians,
    author = {{Villaescusa-Navarro}, Francisco},
    title = "{Pylians: Python libraries for the analysis of numerical simulations}",
    keywords = {Software},
    howpublished = {Astrophysics Source Code Library, record ascl:1811.008},
    year = 2018,
    month = nov,
    eid = {ascl:1811.008},
    pages = {ascl:1811.008},
    archivePrefix = {ascl},
    eprint = {1811.008},
    adsurl = {https://ui.adsabs.harvard.edu/abs/2018ascl.soft11008V},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

CHAPTER

TWENTYTWO

CONTACT

For comments, problems, bugs… etc you can reach me at villaescusa.francisco@gmail.com.