pyCOLA Documentation

Release 1.0

Svetlin Tassev

July 03, 2014

CONTENTS

Author Svetlin Tassev

Version 1.0

Date June 26, 2014

Homepage pyCOLA Homepage

Documentation PDF Documentation

License GPLv3+ License

CHAPTER

ONE

INTRODUCTION

pyCOLA is a multithreaded Python/Cython N-body code, implementing the Comoving Lagrangian Acceleration (COLA) method in the temporal and spatial domains. pyCOLA also implements a novel method to compute second-order cosmological initial conditions for given initial conditions at first-order for arbitrary initial particle configurations (including glass initial conditions, as well as initial conditions having refined subregions).

pyCOLA is based on the following two papers: [temporalCOLA], [spatialCOLA]. We kindly ask you [1] to acknowledge them and their authors in any program or publication in which you use the COLA method in the temporal and/or spatial domains.

The new method for calculating second-order cosmological initial conditions is based on the following paper: (todo: Daniel, let me know what to cite). Again, we kindly ask you to acknowledge that paper and its authors in any program or publication in which you use that method.

pyCOLA requires NumPy, SciPy, pyFFTW, h5py, as well as their respective dependencies. Note that pyFFTW v0.9.2 does not support large arrays, so one needs to install the development version from github, where the bug has been fixed.

Note: All lengthscales are in units of comoving Mpc/h, unless otherwise specified.

Todo

If there is interest in the code (i.e. not only the algorithm), it should be converted to use classes as that will enormously reduce the amount of arguments to be passed around, will make the code more readable, and reduce the chances for introducing bugs. Some of the functions are already converted to using classes in a separate branch, but converting the whole code will take some time.

[1] We do not *require* you, however, as we want pyCOLA to be GPLv3 compatible.

CHAPTER

TWO

PYCOLA MODULES

2.1 Initial conditions

First order initial conditions for pyCOLA can be calculated using either MUSIC [MUSIC], or internally. With MUSIC, however, one can do refinements on a region, which is not supported internally.

The second-order displacement field is generated using a novel algorithm using force evaluations. See the Algorithm section of `ic_2lpt_engine()` for details.

Warning: As of MUSIC rev. 116353436ee6, the second-order displacement field returned by MUSIC gets unphysical large-scale deviations when a refined subvolume is requested (seems to be fine for single grid). Until that problem is fixed, use the function `ic.ic_2lpt()` to get the second order displacements from the first order result. Update: MUSIC received a fix with rev. ed51fcaffee, which supposedly fixes the problem.

2.1.1 Initial displacements at first order

`ic.`**`import_music_snapshot`**(*hdf5_filename*, *boxsize*, *level0='09'*, *level1=None*)

Import a MUSIC snapshot calculated in the ZA.

Arguments:

- `hdf5_filename` – a string. Gives the filename for the HDF5 file, which MUSIC outputs.
- `boxsize` – a float. The size of the full simulation box in Mpc/h.
- `level0` – a two-character string (default: `'09'`). A MUSIC level covering the whole box. With the settings below, it should equal `levelmin` from the MUSIC configuration file for the finest such level.
- `level1` – a two-character string (default: `None`). A fine MUSIC level covering the refined subvolume. With the settings below, it should equal `levelmax` from the MUSIC configuration file for the finest such level.

Return:

- if `level1` is `None`: `(sx,sy,sz)` – a tuple, where `si` is a 3-dim single precision NumPy array containing the `i`-th component (`i` = `x`, `y`, `z`) of the particle displacements today as calculated in the ZA. `si` are the displacements for the `level0` particles.
- if `level1` is not `None`: `(sx,sy,sz,sx_zoom,sy_zoom,sz_zoom,offset)` – a tuple, where:
  - `si` and `si_zoom` are 3-dim single precision NumPy arrays containing the `i`-th component (`i` = `x`, `y`, `z`) of the particle displacements today as calculated in the ZA. `si` are the displacements for the crude level (`level0`) particles; while `si_zoom` are the displacements for the fine level (`level1`) particles in the refined subvolume.

–`offset` – a list of three integers giving the crude-grid index coordinates of the origin of the fine grid.

Note: pyCOLA requires specific values for some keywords in the MUSIC configuration file. Those are:

```
zstart = 0
align_top = yes
use_2LPT = no
format = generic
```

Also, if `level1` is not `None`, pyCOLA assumes that one uses only one (usually the finest) refinement level (`level1`) on the subvolume of interest. Then the following needs to hold:

```
levelmin<levelmax
ref_extent!=1.0,1.0,1.0
```

See the included `ics.conf` for an example.

`ic.`**`ic_za`**(*file_pk*, *boxsize=100.0*, *npart=64*, *init_seed=1234*)

Generates Gaussian initial conditions for cosmological simulations in the Zel'dovich appoximation (ZA) – the first order in Lagrangian Perturbation Theory (LPT).

Arguments:

- `file_pk` – a string. Gives the filename for the plain text file containing the matter power spectrum at redshift zero from CAMB. For an example, see the included `camb_matterpower_z0.dat`.
- `boxsize` – a float (default: `100.0`). Gives the size of the simulation box in Mpc/h.
- `npart` – an integer (default: `64`). The total number of particles is $\texttt{npart}^3$.
- `init_seed` – an integer (default: `1234`). The seed for the random number generator.

Return:

- `(sx,sy,sz)` – a tuple, where $\texttt{s}_\texttt{i}$ is a 3-dim single precision NumPy array containing the `i`-th component (`i` = `x`, `y`, `z`) of the particle displacements today as calculated in the ZA.

Example:

Generate a realization for the displacement field; calculate the rms displacements; and show a projection of one of the components.

```
>>> from ic import ic_za
>>> sx,sy,sz=ic_za('camb_matterpower_z0.dat')
Memory allocation done
Plans created
Power spectrum read.
Randoms done.
Nyquists fixed
sx fft ready
sy fft ready
sz fft ready
>>> ((sx**2+sy**2+sz**2).mean())**0.5/0.7 # O(10) for our universe
10.346006222040094
>>> import matplotlib.pyplot as plt # needs matplotlib to be installed!
>>> plt.imshow(sx.mean(axis=2))
<matplotlib.image.AxesImage object at 0x7fc4603697d0>
>>> plt.show()
```

Algorithm:

Implemented in the usual fft way.

Warning: This function has been tested but not at the level of trusting it for doing research. Use at your own risk.

2.1.2 Initial displacements at second order

`ic.`**`ic_2lpt`**`(`*cellsize*, *sx*, *sy*, *sz*, *sx_zoom=None*, *sy_zoom=None*, *sz_zoom=None*, *boxsize=100.0*, *ngrid_x_lpt=128*, *ngrid_y_lpt=128*, *ngrid_z_lpt=128*, *cellsize_zoom=0*, *offset_zoom=None*, *BBox_in=None*, *growth_2pt_calc=0.05*, *with_4pt_rule=False*, *factor_4pt=2.0*`)`

Given a set of displacements calculated in the ZA at redshift zero, find the corresponding second order displacement. Works with a single grid of particles, as well as with one refined subvolume.

Arguments:

- `cellsize` – a float. The inter-particle spacing in Lagrangian space.
- `sx,sy,sz` – 3-dim NumPy arrays containing the components of the particle displacements today as calculated in the ZA. These particles should cover the whole box. If a refined subvolume is provided, the crude particles which reside inside that subvolume are discarded and replaced with the fine particles.
- `sx_zoom,sy_zoom,sz_zoom` – 3-dim NumPy arrays containing the components of the particle ZA displacements today for a refined subvolume (default: `None`).
- `boxsize` – a float (default: `100.0`). Gives the size of the simulation box in Mpc/h.
- `ngrid_x_lpt,ngrid_y_lpt,ngrid_z_lpt` – integers (default: `128`). Provide the size of the PM grid, which the algorithm uses to calculate the 2LPT displacements.
- `cellsize_zoom` – a float (default: `0`). The inter-particle spacing in Lagrangian space for the refined subvolume, if such is provided. If not, `cellsize_zoom` must be set to zero (default), as that is used as a check for the presence of that subvolume.
- `offset_zoom` – a 3-vector of floats (default: `None`). Gives the physical coordinates of the origin of the refinement region relative to the the origin of the full box.
- `BBox_in` – a 3x2 array of integers (default: `None`). It has the form `[[i0,i1],[j0,j1],[k0,k1]]`, which gives the bounding box for the refinement region in units of the crude particles Lagrangian index. Thus, the particles with displacements `sx|sy|sz[i0:i1,j0:j1,k0:k1]` are replaced with the fine particles with displacements `sx_zoom|sy_zoom|sz_zoom`.
- `growth_2pt_calc` – a float (default: `0.05`). The linear growth factor used internally in the 2LPT calculation. A value of 0.05 gives excellent cross-correlation between the 2LPT field returned by this function, and the 2LPT returned using the usual fft tricks. Yet, some irrelevant short-scale noise is present, which one may decide to filter out. That noise is most probably due to lack of force accuracy for too low `growth_2pt_calc`. Experiment with this value as needed.
- `with_4pt_rule` – a boolean (default: `False`). See `ic.ic_2lpt_engine()`.
- `factor_4pt` – a float (default: `2.0`). See `ic.ic_2lpt_engine()`.

Return:

- If no refined subregion is supplied (indicated by `cellsize_zoom=0`), then return:

  `(sx2,sy2,sz2)` – 3-dim NumPy arrays containing the components of the second order particle displacements today as calculated in 2LPT.

- If a refined subregion is supplied (indicated by `cellsize_zoom>0`), then return:

  `(sx2,sy2,sz2,sx2_zoom,sy2_zoom,sz2_zoom)`

The first three arrays are as above. The last three give the second order displacements today for the particles in the refined subvolume.

Example:

Generate a realization for the displacement field in the ZA; calculate the corresponding second order displacement field; calculate the rms displacements; then show a projection of one of the components.

```
>>> from ic import ic_za,ic_2lpt
>>> sx,sy,sz=ic_za('camb_matterpower_z0.dat',npart=128)
Memory allocation done
Plans created
Power spectrum read.
Randoms done.
Nyquists fixed
sx fft ready
sy fft ready
sz fft ready
>>> sx2,sy2,sz2 = ic_2lpt( 100.0/float(sx.shape[0]),sx,sy,sz,
...                        growth_2pt_calc=0.1)
>>> ((sx**2+sy**2+sz**2).mean())**0.5/0.7    # ~10
11.605451188108798
>>> ((sx2**2+sy2**2+sz2**2).mean())**0.5/0.7 # ~2
2.3447627779313525
>>> import matplotlib.pyplot as plt # needs matplotlib to be installed!
>>> plt.imshow(sx.mean(axis=2))
<matplotlib.image.AxesImage object at 0x7fc4603697d0>
>>> plt.show()
>>> plt.imshow(sy2.mean(axis=2))
<matplotlib.image.AxesImage object at 0x7fc4603697d0>
>>> plt.show()
```

Algorithm:

This function issues a call to `ic.ic_2lpt_engine()`. See the Algorithm section of that function for details.

`ic.`**`ic_2lpt_engine`**(*sx_full, sy_full, sz_full, cellsize, ngrid_x, ngrid_y, ngrid_z, gridcellsize, growth_2pt_calc=0.05, with_4pt_rule=False, factor_4pt=2.0, with_2lpt=False, sx2_full=None, sy2_full=None, sz2_full=None, cellsize_zoom=0, BBox_in=None, sx_full_zoom=None, sy_full_zoom=None, sz_full_zoom=None, sx2_full_zoom=None, sy2_full_zoom=None, sz2_full_zoom=None, offset_zoom=None*)

The same as `ic.ic_2lpt()` above, but calculates the 2LPT displacements for the particles in the COLA volume as generated by same particles displaced according to the 2LPT of the full box. (todo: *expand this!*) In fact, `ic.ic_2lpt()` works by making a call to this function.

Arguments:

- `sx_full,sy_full,sz_full` – 3-dim NumPy arrays containing the components of the particle displacements today as calculated in the ZA of the full box. These particles should cover the whole box. If a refined subvolume is provided, the crude particles which reside inside that subvolume are discarded and replaced with the fine particles.
- `cellsize` – a float. The inter-particle spacing in Lagrangian space.
- `ngrid_x,ngrid_y,ngrid_z` – integers. Provide the size of the PM grid, which the algorithm uses to calculate the 2LPT displacements.
- `gridcellsize` –float. Provide the grid spacing of the PM grid, which the algorithm uses to calculate the 2LPT displacements.

- `growth_2pt_calc` – a float (default: `0.05`). The linear growth factor used internally in the 2LPT calculation. A value of 0.05 gives excellent cross-correlation between the 2LPT field returned by this function, and the 2LPT returned using the usual fft tricks for a 100:math:*mathrm{Mpc}/h* box. Yet, some irrelevant short-scale noise is present, which one may decide to filter out. That noise is probably due to lack of force accuracy for too low `growth_2pt_calc`. Experiment with this value as needed.
- `with_4pt_rule` – a boolean (default: `False`). Whether to use the 4-point force rule to evaluate the ZA and 2LPT displacements in the COLA region. See the Algorithm section below. If set to False, it uses the 2-point force rule.
- `factor_4pt` – a float, different from `1.0` (default: `2.0`). Used for the 4-point force rule. See the Algorithm section below.
- `with_2lpt` – a boolean (default: `False`). Whether the second order displacement field over the full box is provided. One must provide it if the COLA volume is different from the full box. Only if they are the same (as in the case of `ic_2lpt()`) can one set `with_2lpt=False`.
- `sx2_full,sy2_full,sz2_full` – 3-dim NumPy float arrays giving the second order displacement field over the full box. Needs `with_2lpt=True`.
- The rest of the input is as in `ic.ic_2lpt()`, with all LPT quantities provided for the whole box.

Return:

- If no refined subregion is supplied (indicated by `cellsize_zoom=0`), then return:

  `(sx,sy,sz,sx2,sy2,sz2)` – 3-dim NumPy arrays containing the components of the first and second (`s¡2`) order particle displacements today as calculated in 2LPT in the COLA volume.
- If a refined subregion is supplied (indicated by `cellsize_zoom>0`), then return:

`(sx,sy,sz,sx2,sy2,sz2,sx_zoom,sy_zoom,sz_zoom,sx2_zoom,sy2_zoom,sz2_zoom)`

The first 6 arrays are as above. The last 6 give the second order displacements today for the particles in the refined subvolume of the COLA volume.

Algorithm:

The first-order and second-order displacements, $\boldsymbol{s}^{(1)}_{\mathrm{COLA}}$ and $\boldsymbol{s}^{(2)}_{\mathrm{COLA}}$, in the COLA volume at redshift zero are calculated according to the following 2-pt or 4-pt (denoted by subscript) equations:

$$\begin{aligned}
\boldsymbol{s}^{(1)}_{\mathrm{COLA,2pt}} &= -\frac{1}{2g}\left[\boldsymbol{F}(g,\beta g^2)-\boldsymbol{F}(-g,\beta g^2)\right] && (2.1)\\
\boldsymbol{s}^{(2)}_{\mathrm{COLA,2pt}} &= -\frac{\alpha}{2g^2}\left[\boldsymbol{F}(g,\beta g^2)+\boldsymbol{F}(-g,\beta g^2)\right] && (2.2)\\
\boldsymbol{s}^{(1)}_{\mathrm{COLA,4pt}} &= -\frac{1}{2g}\frac{a^2}{a^2-1}\Bigg[\boldsymbol{F}(g,\beta g^2)-\boldsymbol{F}(-g,\beta g^2)- && (2.3)\\
&\qquad -\frac{1}{a^3}\Bigg(\boldsymbol{F}\left(ag,\beta a^2g^2\right)-\boldsymbol{F}\left(-ag,\beta a^2g^2\right)\Bigg)\Bigg] && (2.4)\\
\boldsymbol{s}^{(2)}_{\mathrm{COLA,4pt}} &= -\frac{\alpha}{2g^2}\frac{a^2}{a^2-1}\Bigg[\boldsymbol{F}(g,\beta g^2)+\boldsymbol{F}(-g,\beta g^2)- && (2.5)\\
&\qquad -\frac{1}{a^4}\Bigg(\boldsymbol{F}\left(ag,\beta a^2g^2\right)+\boldsymbol{F}\left(-ag,\beta a^2g^2\right)\Bigg)\Bigg] && (2.6)
\end{aligned}$$

where:

$a =$ `factor_4pt`

$g =$ `growth_2pt_calc`

if `with_2lpt` then:

$\beta = 1$ and $\alpha = (3/10)\Omega_m^{1/143}$

else:

$\beta = 0$ and $\alpha = (3/7)\Omega_m^{1/143}$

The factors of $\Omega_m^{1/143}$ (Ω_m being the matter density today) are needed to rescale the second order displacements to matter domination and are correct to $\mathcal{O}(\max(10^{-4}, g^3/143))$ in ΛCDM. The force $\boldsymbol{F}(g_1, g_2)$ is given by:

$$\boldsymbol{F}(g_1, g_2) = \boldsymbol{\nabla}\nabla^{-2}\delta\left[g_1\boldsymbol{s}^{(1)}_{\rm full} + g_2\Omega_m^{-1/143}\boldsymbol{s}^{(2)}_{\rm full}\right] \tag{2.7}$$

where $\delta[\boldsymbol{s}]$ is the cloud-in-cell fractional overdensity calculated from a grid of particles displaced by the input displacement vector field $\boldsymbol{s}$. Above, $\boldsymbol{s}^{(1)}_{\rm full}/\boldsymbol{s}^{(2)}_{\rm full}$ are the input first/second-order input displacement fields calculated in the full box at redshift zero.

It is important to note that implicitly for each particle at Lagrangian position $\boldsymbol{q}$, the force $\boldsymbol{F}(g_1, g_2)$ is evaluated at the corresponding Eulerian position: $\boldsymbol{q} + g_1\boldsymbol{s}^{(1)}_{\rm full} + g_2\Omega_m^{-1/143}\boldsymbol{s}^{(2)}_{\rm full}$.

As noted above, `with_2lpt=False` is only allowed if the COLA volume covers the full box volume. In that case, $\boldsymbol{s}^{(2)}_{\rm full}$ is not needed as input since $\beta = 0$. Instead, the output $\boldsymbol{s}^{(2)}_{\rm COLA}$ can be used as a good approximation to $\boldsymbol{s}^{(2)}_{\rm full}$. This fact is used in `ic.ic_2lpt()` to calculate $\boldsymbol{s}^{(2)}_{\rm full}$ from $\boldsymbol{s}^{(1)}_{\rm full}$.

Note: If `with_4pt_rule=False`, then the first/second order displacements receive corrections at third/fourth order. If `with_4pt_rule=True`, then those corrections are fifth/sixth order. However, when using the 4-point rule instead of the 2-point rule, one must make two more force evaluations at a slightly different growth factor given by `growth_2pt_calc*factor_4pt`. Since the code is single precision and is using a simple PM grid to evaluate forces, one cannot make `factor_4pt` and `growth_2pt_calc` too small due to noise issues. Thus, when comparing the 2-pt and 4-pt rule, we should assume `factor_4pt>1`. And again due to numerical precision issues, one cannot choose `factor_4pt` to be too close to one; hence, the default value of `2.0`.

Therefore, as the higher order corrections for the 4-pt rule are proportional to powers of `growth_2pt_calc*factor_4pt`, one may be better off using the 2-pt rule (the default) in this particular implementation. Yet for codes where force accuracy is not an issue, one may consider using the 4-pt rule. Thus, its inclusion in this code is mostly done as an illustration.

2.1.3 Obtaining the Eulerian positions

`ic.`**`initial_positions`**(*sx, sy, sz, sx2, sy2, sz2, cellsize, growth_factor, growth_factor_2lpt, ngrid_x, ngrid_y, ngrid_z, gridcellsize, offset=[0.0, 0.0, 0.0]*)

Add the Lagrangian particle position to the 2LPT displacement to obtain the Eulerian position. Periodic boundary conditions are assumed.

Arguments:

- `sx,sy,sz` – 3-dim NumPy arrays containing the components of the particle displacements today as calculated in the ZA.
- `sx2,sy2,sz2` – 3-dim NumPy arrays containing the components of the second order particle displacements today as calculated in 2LPT.
- `cellsize` – a float. The inter particle spacing in Lagrangian space.
- `growth_factor` – a float. The linear growth factor for the redshift for which the Eulerian positions are requested.
- `growth_factor_2lpt` – a float. The second order growth factor for the redshift for which the Eulerian positions are requested.

- `ngrid_x`, `ngrid_y`, `ngrid_z` – integers. The grid size of the box. Only used together with `gridcellsize` below to find the physical size of the box, which is needed to apply the periodic boundary conditions.
- `gridcellsize` – a float. The grid spacing of the box.
- `offset` – a list of three floats (default: `[0.0,0.0,0.0]`). Offset the Eulerian particle positions by this amount. Useful for placing refined subregions at their proper locations inside a bigger box.

Return:

- `(px,py,pz)` – a tuple, where $\mathrm{p_i}$ is a 3-dim single precision NumPy array containing the `i`-th component (`i` = `x`, `y`, `z`) of the particle Eulerian position.

Example:

In this example we generate the initial conditions in 2LPT, and then plot a slice through the 2LPT realization at redshift of zero.

```
>>> from ic import ic_za,ic_2lpt,initial_positions
>>> sx,sy,sz=ic_za('camb_matterpower_z0.dat',npart=128)
Memory allocation done
Plans created
Power spectrum read.
Randoms done.
Nyquists fixed
sx fft ready
sy fft ready
sz fft ready
>>> sx2,sy2,sz2 = ic_2lpt( 100.0/float(sx.shape[0]),sx,sy,sz,
...                        growth_2pt_calc=0.1)
>>> px,py,pz = initial_positions(sx,sy,sz,sx2,sy2,sz2,100./128.,1.0,1.0,
...             1, # only ngrid_i*gridcellsize=boxsize is relevant here
...             1,
...             1,
...             100.0)
>>> import matplotlib.pyplot as plt # needs matplotlib to be installed
>>> import numpy as np
>>> ind=np.where(pz<3)
>>> px_slice=px[ind]
>>> py_slice=py[ind]
>>> plt.figure(figsize=(10,10))
<matplotlib.figure.Figure object at 0x7f21044d2e10>
>>> plt.scatter(px_slice,py_slice,marker='.',alpha=0.03,color='r')
<matplotlib.collections.PathCollection object at 0x7f2102cd3290>
>>> plt.show()
```

2.2 Growth factors

growth.**growth_factor_solution**(*Om*, *Ol*)

Calculate the linear growth factor evolution for a given cosmology.

Arguments:

- `Om` – a float, giving the matter density, Ω_m, today.
- `Ol` – a float, giving the vacuum density, Ω_Λ, today.

Return:

- an $n \times 3$ array containing $[a_i, D(a_i), T[D](a_i)]$ for $i = 1 \ldots n$ in order of increasing scale factor a. Here, the linear growth factor is given by $D(a)$, while $T[D](a)$ is given by equation (A.1) of [temporalCOLA]. These arrays can be further interpolated if needed.

growth.**growth_2lpt** (*a*, *d*, *Om*)

Return the second order growth factor for a given scale factor and respective linear growth factor. One needs to precompute the latter. ΛCDM is assumed for this calculation.

Arguments:

- `a` – a float, giving the scale factor.
- `d` – a float, giving the linear growth factor at a.
- `Om` – a float, giving the matter density, Ω_m, today.

Return:

- A float giving the second order growth factor.

Example:

```
>>> Om=0.275
>>> Ol=1.0-Om
>>> from growth import growth_factor_solution,growth_2lpt
>>> darr=growth_factor_solution(Om,Ol)
>>> from scipy import interpolate
>>> growth = interpolate.interp1d(darr[:,0].tolist(),darr[:,1].tolist(),
...                               kind='linear')
>>> a=0.3
>>> d=growth(a)
>>> growth_2lpt(a,d,Om)/d/d
0.99148941733187124
```

growth.**d_growth2** (*a*, *d*, *Om*, *Ol*)

Return $T[D_2](a)$ for the second order growth factor, D_2, for a given scale factor and respective linear growth factor. Here T is given by equation (A.1) of [temporalCOLA]. One needs to precompute the linear growth factor. ΛCDM is assumed for this calculation.

Arguments:

- `a` – a float, giving the scale factor.
- `d` – a float, giving the linear growth factor at a.
- `Om` – a float, giving the matter density, Ω_m, today.

Return:

- A float giving $T[D_2](a)$.

2.3 Cloud-in-Cell

cic.**CICDeposit_3** (*ndarray sx*, *ndarray sy*, *ndarray sz*, *ndarray sx2*, *ndarray sy2*, *ndarray sz2*, *ndarray field*, *float32_t cellsize*, *float32_t gridcellsize*, *int32_t add_lagrangian_position*, *float32_t growth_factor*, *float32_t growth_factor2*, *ndarray BBox*, *ndarray offset*, *int32_t periodic*)

Do a cloud-in-cell (CiC) assignment.

Arguments:

- `sx,sy,sz` – 3-dim float32 arrays. If `add_lagrangian_position=0`, these arrays contain the three components of the particle Eulerian positions. If `add_lagrangian_position=1`, they contain the particle linear displacemens today.
- `sx2,sy2,sz2` – 3-dim float32 arrays. Contain the components of the particle second-order displacemens today. Not used if `add_lagrangian_position=0`.
- `field` – 3-dim float32 array. Contains the density field after the CiC assignment.
- `cellsize` – float32. The interparticle spacing.
- `gridcellsize` – float32. The grid spacing.
- `add_lagrangian_position` – int32. See the arguments above.
- `growth_factor` – float32. The linear growth factor by which to multiply the `si` arrays when `add_lagrangian_position=1`. Not used when `add_lagrangian_position=0`.
- `growth_factor2` – float32. The second-order growth factor by which to multiply the `si2` arrays when `add_lagrangian_position=1`. Not used when `add_lagrangian_position=0`.
- `BBox` – 2-dim int32 array. The bounding box of a Lagrangian region which must be omitted in the CiC assignment. If we denote its elements by `[[i0,i1],[j0,j1],[k0,k1]]`, then the particles with positions/displacements `sx|sy|sz[i0:i1,j0:j1,k0:k1]` are skipped in the CiC. Used to carve a box inside a grid of crude particles, to be filled later with fine particles.
- `offset` – 1-dim float32 array. The 3-vector offset of the particles, in units of Mpc/h. Not used when `add_lagrangian_position=0`.
- `periodic` – int32. If `periodic=1`, use periodic boundary conditions.

Result:

- `field` is modified by adding to it the CiC assigned particles.

Warning: The CiC is done on `field` without first setting it to a default value. If you need to, fill it with zeros before calling this function.

2.4 Potential

`potential.`**`initialize_density`**(*ngrid_x*, *ngrid_y*, *ngrid_z*)

Initialize the PM grid and its forward and inverse in-place Fourier transforms. We use pyFFTW, which issues calls to the fftw library to create the plans for the FFT.

Arguments:

- `ngrid_x,ngrid_y,ngrid_z` – integers, giving the size of the PM grid.

Return:

- `density` – a properly aligned 3-dim float32 array.
- `density_k` – a view of `density` as a 3-dim complex64 array. After executing the forward fft plan, `density_k` contains the in-place fft'd `density`.
- `den_fft` – instance of the FFTW class for computing the forward fft, which fft's `density` to give `density_k`. Creating the instance is equivalent to creating a fftw plan. Calling the instance, executes the plan.
- `den_ifft` – instance of the FFTW class for computing the inverse fft, which ifft's `density_k` to give back `density` (up to normalization).

`potential.`**`get_phi`**(*ndarray denphi*, *ndarray den_k*, *den_fft*, *phi_ifft*, *int32_t ngrid_x*, *int32_t ngrid_y*, *int32_t ngrid_z*, *float32_t gridcellsize*)

Calculate the potential sourced by a given density field. Periodic boundary conditions are assumed.

Arguments:

- `denphi,den_k,den_fft,phi_ifft` – these arrays and classes are the output from a single call to `potential.initialize_density()`:

  ```
  denphi,den_k,den_fft,phi_ifft = initialize_density(ngrid_x,ngrid_y,ngrid_z)
  ```

  The array `denphi` should then be assigned the values of the density field, and then fed as an input to this function. It is overwritten with the values of the potential on exit.

- `ngrid_x, ngrid_y, ngrid_z` – int32. The size of `denphi`.
- `gridcellsize` – float32. Grid spacing of the PM grid in physical units.

Result:

- `denphi` contains the potential on exit.

Algorithm:

Convolve the input density with the $-1/k^2$ kernel.

2.5 Accelerations

`acceleration.`**`grad_phi`**(*ndarray sx*, *ndarray sy*, *ndarray sz*, *ndarray sx2*, *ndarray sy2*, *ndarray sz2*, *ndarray velx*, *ndarray vely*, *ndarray velz*, *int32_t npart_x*, *int32_t npart_y*, *int32_t npart_z*, *ndarray field*, *int32_t ngrid_x*, *int32_t ngrid_y*, *int32_t ngrid_z*, *float32_t cellsize*, *float32_t gridcellsize*, *float32_t growth*, *float32_t growth2*, *ndarray offset*)

Calculate the gradient of a potential by issuing a call to `acceleration.grad_phi_engine()`. Arguments are the same as in `acceleration.grad_phi_engine()` but internally it sets:

```
add_lagrangian_position=1
beta1=1
beta2=0
```

And `vel`$_{\rm i}$ is set to zero first, i.e. $\boldsymbol{v}_{\rm in} = 0$.

`acceleration.`**`grad_phi_engine`**(*ndarray posx*, *ndarray posy*, *ndarray posz*, *ndarray velx*, *ndarray vely*, *ndarray velz*, *ndarray sx*, *ndarray sy*, *ndarray sz*, *ndarray sx2*, *ndarray sy2*, *ndarray sz2*, *float32_t beta1*, *float32_t beta2*, *int32_t npart_x*, *int32_t npart_y*, *int32_t npart_z*, *ndarray field*, *int32_t ngrid_x*, *int32_t ngrid_y*, *int32_t ngrid_z*, *float32_t cellsize*, *float32_t gridcellsize*, *float32_t growth*, *float32_t growth2*, *ndarray offset*, *int32_t add_lagrangian_position*)

Calculate particle accelerations using a finite difference scheme to save memory. In particular, the function evaluates the following equation for each particle:

$$\boldsymbol{v}_{\rm out} = \boldsymbol{v}_{\rm in} + \beta_1 \boldsymbol{\nabla}\phi + \beta_2 \left(g_1 \boldsymbol{s}^{(1)} + g_1 \boldsymbol{s}^{(2)} \right) \tag{2.8}$$

If `add_lagrangian_position=0`, then $\boldsymbol{\nabla}\phi$ is evaluated at position `pos`$_{\rm i}$ for each particle. If `add_lagrangian_position=1`, the gradient is evaluated at the particle position given by its Lagrangian position plus a displacement $g_1\boldsymbol{s}^{(1)} + g_1\boldsymbol{s}^{(2)}$+`offset`. In the latter case, periodic boundary conditions are assumed.

Arguments:

- posx,posy,posz – 3-dim float32 arrays. Not used when add_lagrangian_position=1. See above.
- velx,vely,velz – 3-dim float32 arrays, containing the components of $\boldsymbol{v}_{\rm in}$ above. Overwritten on output to contain $\boldsymbol{v}_{\rm out}$.
- sx,sy,sz – 3-dim float32 arrays, containing the components of $\boldsymbol{s}^{(1)}$ above.
- sx2,sy2,sz2 – 3-dim float32 arrays, containing the components of $\boldsymbol{s}^{(2)}$ above.
- beta1,beta2 – float32. Equal β_1 and β_2 above, respectively.
- npart_x,npart_y,npart_z – int32, giving the size of the input particle arrays (e.g. sx).
- field – 3-dim float32 array, containing the potential ϕ above.
- ngrid_x,ngrid_y,ngrid_z – int32, giving the size of the field array.
- cellsize – float32. The interparticle spacing in physical units.
- gridcellsize – float32. The grid spacing in physical units.
- growth, growth2 – float32. Equal g_1 and :math:g_2‘ above, respectively.
- offset – 1-dim float32 array. Not used when add_lagrangian_position=0. See above.
- add_lagrangian_position – int32. See above.

Result:

- The arrays velx,vely,velz are updated according to the equation above.

Algorithm:

Use a 4-point finite difference scheme [1] combined with a bi-linear interpolation in the orthogonal directions. The coefficients for the 4-pt finite difference are derived below with SymPy. Use this piece of code to generate coefficients for higher/lower-order difference schemes as needed:

```
>>> from sympy import *
>>> a,da,dx,x=var('a da dx x')
>>> fa,fp1,fm1,fp2=var('fa fp1 fm1 fp2')
>>> d1,d2,d3=var('d1 d2 d3')
>>> f=Function('f')
>>> deriv_dict={f(a):fa,Subs(diff(f(x),x),(x,),(a,)) : d1,
...                       Subs(diff(f(x),x,x),(x,),(a,)):d2,
...                       Subs(diff(f(x),x,x,x),(x,),(a,)):d3}
>>> ftaylorTemp=series(f(x),x,a,4)#Call this only once if using SymPy<=0.7.5.
>>>                               #Repeated calls are buggy.
>>>                               #Problem fixed on github.
>>> ftaylor=(ftaylorTemp.xreplace(deriv_dict)).removeO()
>>> #ftaylor=ftaylor.subs({x:x-a}) # Uncomment this if using SymPy<=0.7.5.
>>>                                # Fixed on github.
>>> sol=solve([ftaylor.subs({x:a+da})-fp1, ftaylor.subs({x:a+2*da})-fp2,
...            ftaylor.subs({x:a-da})-fm1],[d1,d2,d3],solution_dict=True)
>>> res = diff(ftaylor,x).subs({x:a+dx*da}).subs(deriv_dict).subs(sol)
>>> (6*da*res).expand().collect(fm1).collect(fp2).collect(fp1).collect(fa)
f_0*(9*dx**2 - 12*dx - 3) + fm1*(-3*dx**2 + 6*dx - 2) +
fp1*(-9*dx**2 + 6*dx + 6) + fp2*(3*dx**2 - 1)
```

[1] Not to be confused with the 4-pt calculation of the second-order initial conditions.

2.6 Evolution

`evolve.evolve`(*cellsize*, *sx_full*, *sy_full*, *sz_full*, *sx2_full*, *sy2_full*, *sz2_full*, *FULL=False*, *cellsize_zoom=0*, *sx_full_zoom=None*, *sy_full_zoom=None*, *sz_full_zoom=None*, *sx2_full_zoom=None*, *sy2_full_zoom=None*, *sz2_full_zoom=None*, *offset_zoom=None*, *BBox_in=None*, *ngrid_x=None*, *ngrid_y=None*, *ngrid_z=None*, *gridcellsize=None*, *ngrid_x_lpt=None*, *ngrid_y_lpt=None*, *ngrid_z_lpt=None*, *gridcellsize_lpt=None*, *Om=0.274*, *Ol=0.726*, *a_initial=0.06666666666666667*, *a_final=1.0*, *n_steps=15*, *nCola=-2.5*, *save_to_file=False*, *file_npz_out='tmp.npz'*)

Evolve a set of initial conditions forward in time using the COLA method in both the spatial and temporal domains.

Arguments:

- `cellsize` – a float. The inter-particle spacing in Lagrangian space.
- `sx_full,sy_full,sz_full` – 3-dim NumPy float32 arrays containing the components of the particle displacements today as calculated in the ZA in the full box. These particles should cover the COLA volume only. If a refined subvolume is provided, these crude particles which reside inside that subvolume are discarded and replaced with the fine particles. There arrays are overwritten.
- `sx2_full,sy2_full,sz2_full` – same as above but for the second order displacement field.
- `FULL` – a boolean (default: `False`). If True, it indicates to the code that the COLA volume covers the full box. In that case, LPT in the COLA volume is not calculated, as that matches the LPT in the full box.
- `cellsize_zoom` – a float (default: `0`). The inter-particle spacing in Lagrangian space for the refined subvolume, if such is provided. If not, `cellsize_zoom` must be set to zero (default), as that is used as a check for the presence of that subvolume.
- `s*_full_zoom,s*2_full_zoom` – same as without `_zoom` above, but for the refined region (default: `None`).
- `offset_zoom` – a 3-vector of floats (default: `None`). Gives the physical coordinates of the origin of the refinement region relative to the the origin of the full box.
- `BBox_in` – a 3x2 array of integers (default: `None`). It has the form `[[i0,i1],[j0,j1],[k0,k1]]`, which gives the bounding box for the refinement region in units of the crude particles Lagrangian index. Thus, the particles with displacements `sx_full|sy_full|sz_full[i0:i1,j0:j1,k0:k1]` are replaced with the fine particles with displacements `sx_full_zoom|sy_full_zoom|sz_full_zoom`.
- `ngrid_x,ngrid_y,ngrid_z` – integers (default: `None`). Provide the size of the PM grid, which the algorithm uses to calculate the forces for the Kicks.
- `gridcellsize` –float (default: `None`). Provide the grid spacing of the PM grid, which the algorithm uses to calculate the forces for the Kicks.
- `ngrid_x_lpt,ngrid_y_lpt,ngrid_z_lpt,gridcellsize_lpt` – the same as without `_lpt` above but for calculating the LPT displacements in the COLA volume. These better match their counterparts above for the force calculation, as mismatches often lead to unexpected non-cancellations and artifacts.
- `Om` – a float (default: `0.274`), giving the matter density, Ω_m, today.
- `Ol` – a float (default: `1.-0.274`), giving the vacuum density, Ω_Λ, today.
- `a_initial` – a float (default: `1./15.`). The initial scale factor from which to start the COLA evolution. This better be near `1/n_steps`.
- `a_final` – a float (default: `1.0`). The final scale factor for the COLA evolution.

- `n_steps` – an integer (default: `15`). The total number of timesteps which the COLA algorithm should make.
- `nCola` – a float (default: `-2.5`). The spectral index for the time-domain COLA. Sane values lie in the range `(-4,3.5)`. Cannot be `0`, but of course can be near `0` (say `0.001`). See Section A.3 of [temporalCOLA].
- `save_to_file` – a boolean (default: `False`). Whether to save the final snapshot to file.
- `file_npz_out` – a string (default: `'tmp.npz'`), giving the filename for the `.npz` SciPy file, in which to save the snapshot. See the source file for what is actually saved.

Return:

- `px,py,pz` – 3-dim float32 arrays containing the components of the particle positions inside the COLA volume.
- `vx,vy,vz` – 3-dim float32 arrays containing the components of the particle velocities, $\boldsymbol{v}$. Velocities are in units of Mpc/h and are calculated according to:

$$\boldsymbol{v} \equiv \frac{1}{a\,H(a)}\frac{d\boldsymbol{x}}{d\eta} \tag{2.9}$$

  where η is conformal time; a is the final scale factor `a_final`; $H(a)$ is the Hubble parameter; $\boldsymbol{x}$ is the comoving position. This definition allows calculating redshift-space positions trivial: one simply has to add the line-of-sight velocity to the particle position.

2.7 Auxiliary

`box_smooth.`**`box_smooth`**(*ndarray arr*, *ndarray arr1*)

Do a 3x3x3 boxcar smoothing.

Arguments:

- `arr` – a 3-dim float32 array serving as input.
- `arr1` – a 3-dim float32 array serving as output.

Note: This should really be replaced with a Gaussian smoothing, so that one can change the amount of smoothing. Gaussian smoothing can be trivially implemented by modifying `potential.get_phi()` as indicated in the source file of that function. Not done here as this worked well enough for the paper.

`aux.`**`boundaries`**(*boxsize*, *level*, *level_zoom*, *NPART_zoom*, *offset_from_code*, *cut_from_sides*, *gridscale*)

Calculate bounding boxes, fine grid offsets and other quantities useful when dealing with a MUSIC snapshot.

Arguments:

- The first three arguments must be set to the following parameters from the MUSIC configuration file:

```
[boundaries() argument]  [parameter from MUSIC .conf]    [type]
boxsize                  =          boxlength             float
level                    =          levelmin                int
level_zoom               =          levelmax                int
```

  With the included MUSIC configuration file, `ics.conf`, the above three arguments take the values: 100.0, 9, 10.

- `NPART_zoom` – list of three integers, giving the size of the fine grid.

- `offset_from_code` – list of three integers, giving the crude-grid index coordinates of the origin of the fine grid.
- The last two parameters define the COLA volume and the size of the PM grid:
  - `cut_from_sides` – a list of two integers, call them a, b. Then, the Lagrangian COLA volume in terms of the indices of the particle displacements arrays, `si`, at level `level` (the one covering the full box) is given by the following array slice:

    ```
    si[a:2**level-b,a:2**level-b,a:2**level-b]
    ```

  - `gridscale` – an integer. Sets the size of the PM grid in each dimension to `gridscale` times the particle number at level `level` in that dimension within the COLA volume.

Return:

- `BBox_in,offset_zoom,cellsize,cellsize_zoom` – the same as in `ic.ic_2lpt()` but the first two give the bounding box and offset of the refinement region with respect to the COLA volume.
- `offset_index` – a 3-vector of integers. The same as `offset_zoom` but in units of the crude-particle index.
- `BBox_out` – a 3x2 array of integers. Gives the bounding box at level `level` that resides in the COLA volume. It equals `[[a,2**level-b],[a,2**level-b],[a,2**level-b]]` (see the description of the argument `cut_from_sides` above).
- `BBox_out_zoom` – the same as `BBox_out` but for the fine particles (as level `level_zoom`) included in the COLA volume.
- `ngrid_x, ngrid_y, ngrid_z` – integers. The size of the PM grid used in the COLA volume.
- `gridcellsize` – a float. The PM grid spacing in Mpc/h.

Example: For example usage, see the worked out example in (todo).

CHAPTER

THREE

WORKED-OUT EXAMPLE

The following example is contained in `example.py`. It reads in a MUSIC-generades initial conditions at first order; calculates the second order initial displacements on the full box; then runs COLA and finally outputs a figure containing a density slice. The script needs matplotlib to be installed.

To run the script, first generate MUSIC initial conditions with the included configuration file:

```
MUSIC ics.conf
```

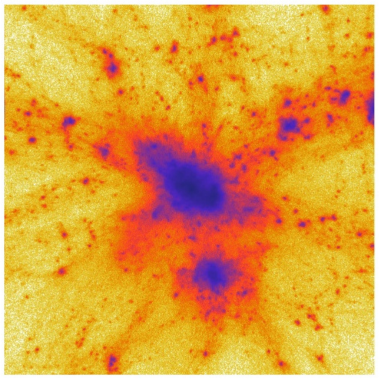
Then change the variable `music_file` below to point to the MUSIC snapshot, and then execute the example script by issuing:

```
python ./example.py
```

This example script was used with minor modifications in making the figures for the paper. It fits comfortably on 24GB ram. If that is not available, decreasing `gridscale` to 1, reduces ram consumption to 11GB at the cost of reducing force resolution and producing artifacts.

Most of the time, the script spends on calculating the second-order displacement field for the whole box, i.e. not on the COLA calculation. On the 4-core laptop the calculations for the paper were performed, the COLA calculation itself takes about 40 seconds.

The script produces the following figure:



```
import numpy as np
import matplotlib.pyplot as plt
from aux import boundaries
from ic import ic_2lpt,import_music_snapshot
from evolve import evolve
from cic import CICDeposit_3
from potential import initialize_density
```

```
# Set up the parameters from the MUSIC ic snapshot:
music_file="/media/stuff/ohahn-music-116353436ee6/ic_za.hdf5" # CHANGE!

# Set up according to instructions for
# aux.boundaries()

boxsize=100.0 # in Mpc/h
level=9
level_zoom=10
gridscale=3

# Set up according to instructions for
# ic.import_music_snapshot()
level0='09' # should match level above
level1='10' # should match level_zoom above


# Set how much to cut from the sides of the full box.
# This makes the COLA box to be of the following size in Mpc/h:
# (2.**level-(cut_from_sides[0]+cut_from_sides[1]))/2.**level*boxsize

# This is the full box. Set FULL=True in evolve() below
#cut_from_sides=[0,0]# 100Mpc/h.
#
# These are the interesting cases:
#cut_from_sides=[64,64]# 75Mpc/h
#cut_from_sides=[128,128] # 50Mpc/h
cut_from_sides=[192,192]  # 25Mpc/h



sx_full1, sy_full1, sz_full1, sx_full_zoom1, sy_full_zoom1, \
    sz_full_zoom1, offset_from_code \
    = import_music_snapshot(music_file, \
                            boxsize,level0=level0,level1=level1)

NPART_zoom=list(sx_full_zoom1.shape)


print "Starting 2LPT on full box."

#Get bounding boxes for full box with 1 refinement level for MUSIC.
BBox_in, offset_zoom, cellsize, cellsize_zoom, \
    offset_index, BBox_out, BBox_out_zoom, \
    ngrid_x, ngrid_y, ngrid_z, gridcellsize  \
    = boundaries(boxsize, level, level_zoom, \
                 NPART_zoom, offset_from_code, [0,0], gridscale)

sx2_full1, sy2_full1, sz2_full1,  sx2_full_zoom1, \
    sy2_full_zoom1, sz2_full_zoom1 \
    = ic_2lpt(
        cellsize,
        sx_full1 ,
        sy_full1 ,
        sz_full1 ,

        cellsize_zoom=cellsize_zoom,
        sx_zoom = sx_full_zoom1,
        sy_zoom = sy_full_zoom1,
        sz_zoom = sz_full_zoom1,

        boxsize=100.00,
        ngrid_x_lpt=ngrid_x,ngrid_y_lpt=ngrid_y,ngrid_z_lpt=ngrid_z,

        offset_zoom=offset_zoom,BBox_in=BBox_in)




#Get bounding boxes for the COLA box with 1 refinement level for MUSIC.
BBox_in, offset_zoom, cellsize, cellsize_zoom, \
    offset_index, BBox_out, BBox_out_zoom, \
    ngrid_x, ngrid_y, ngrid_z, gridcellsize \
    = boundaries(
        boxsize, level, level_zoom, \
        NPART_zoom, offset_from_code, cut_from_sides, gridscale)

# Trim full-box displacement fields down to COLA volume.
sx_full       =       sx_full1[BBox_out[0,0]:BBox_out[0,1],  \
                               BBox_out[1,0]:BBox_out[1,1],  \
                               BBox_out[2,0]:BBox_out[2,1]]
sy_full       =       sy_full1[BBox_out[0,0]:BBox_out[0,1],
                               BBox_out[1,0]:BBox_out[1,1],  \
                               BBox_out[2,0]:BBox_out[2,1]]
sz_full       =       sz_full1[BBox_out[0,0]:BBox_out[0,1],  \
                               BBox_out[1,0]:BBox_out[1,1],  \
                               BBox_out[2,0]:BBox_out[2,1]]
sx_full_zoom  =  sx_full_zoom1[BBox_out_zoom[0,0]:BBox_out_zoom[0,1],  \
                               BBox_out_zoom[1,0]:BBox_out_zoom[1,1],  \
                               BBox_out_zoom[2,0]:BBox_out_zoom[2,1]]
sy_full_zoom  =  sy_full_zoom1[BBox_out_zoom[0,0]:BBox_out_zoom[0,1],  \
                               BBox_out_zoom[1,0]:BBox_out_zoom[1,1],  \
                               BBox_out_zoom[2,0]:BBox_out_zoom[2,1]]
sz_full_zoom  =  sz_full_zoom1[BBox_out_zoom[0,0]:BBox_out_zoom[0,1],  \
                               BBox_out_zoom[1,0]:BBox_out_zoom[1,1],  \
                               BBox_out_zoom[2,0]:BBox_out_zoom[2,1]]
del sx_full1, sy_full1, sz_full1, sx_full_zoom1, sy_full_zoom1, sz_full_zoom1

sx2_full       =       sx2_full1[BBox_out[0,0]:BBox_out[0,1],  \
                                 BBox_out[1,0]:BBox_out[1,1],  \
                                 BBox_out[2,0]:BBox_out[2,1]]
sy2_full       =       sy2_full1[BBox_out[0,0]:BBox_out[0,1],  \
                                 BBox_out[1,0]:BBox_out[1,1],  \
                                  BBox_out[2,0]:BBox_out[2,1]]
sz2_full       =       sz2_full1[BBox_out[0,0]:BBox_out[0,1],  \
                                 BBox_out[1,0]:BBox_out[1,1],  \
                                 BBox_out[2,0]:BBox_out[2,1]]
sx2_full_zoom  =  sx2_full_zoom1[BBox_out_zoom[0,0]:BBox_out_zoom[0,1],  \
                                 BBox_out_zoom[1,0]:BBox_out_zoom[1,1],  \
                                 BBox_out_zoom[2,0]:BBox_out_zoom[2,1]]
sy2_full_zoom  =  sy2_full_zoom1[BBox_out_zoom[0,0]:BBox_out_zoom[0,1],  \
                                 BBox_out_zoom[1,0]:BBox_out_zoom[1,1],  \
                                 BBox_out_zoom[2,0]:BBox_out_zoom[2,1]]
sz2_full_zoom  =  sz2_full_zoom1[BBox_out_zoom[0,0]:BBox_out_zoom[0,1],  \
                                 BBox_out_zoom[1,0]:BBox_out_zoom[1,1],  \
                            BBox_out_zoom[2,0]:BBox_out_zoom[2,1]]
del sx2_full1, sy2_full1, sz2_full1, sx2_full_zoom1, sy2_full_zoom1, sz2_full_zoom1


print "2LPT on full box is done."
print "Starting COLA!"



px, py, pz, vx, vy, vz, \
px_zoom, py_zoom, pz_zoom, vx_zoom, vy_zoom, vz_zoom \
= evolve(
    cellsize,
    sx_full, sy_full, sz_full,
    sx2_full, sy2_full, sz2_full,
    FULL=False,

    cellsize_zoom=cellsize_zoom,
    sx_full_zoom  = sx_full_zoom ,
    sy_full_zoom  = sy_full_zoom ,
    sz_full_zoom  = sz_full_zoom ,
    sx2_full_zoom = sx2_full_zoom,
    sy2_full_zoom = sy2_full_zoom,
    sz2_full_zoom = sz2_full_zoom,

    offset_zoom=offset_zoom,
    BBox_in=BBox_in,

    ngrid_x=ngrid_x,
    ngrid_y=ngrid_y,
    ngrid_z=ngrid_z,
    gridcellsize=gridcellsize,

    ngrid_x_lpt=ngrid_x,
    ngrid_y_lpt=ngrid_y,
    ngrid_z_lpt=ngrid_z,
    gridcellsize_lpt=gridcellsize,

    a_final=1.,
    a_initial=1./10.,
    n_steps=10,

    save_to_file=False,  # set this to True to output the snapshot to a file
    file_npz_out='tmp.npz',
    )

del vx_zoom,vy_zoom,vz_zoom
del vx,vy,vz



print "Making a figure ..."
# grid size for figure array
ngrid=2*512
# physical size of figure array
cutsize=12.0#Mpc/h

# offset vector [Mpc/h]:
```

```
com=[ 1.30208333,  1.10677083,  0.944]
com[0] += offset_zoom[0]+cellsize_zoom * \
    (BBox_out_zoom[0,1]-BBox_out_zoom[0,0])/2.0-cutsize/2.0
com[1] += offset_zoom[1]+cellsize_zoom * \
    (BBox_out_zoom[1,1]-BBox_out_zoom[1,0])/2.0-cutsize/2.0
com[2] += offset_zoom[2]+cellsize_zoom * \
    (BBox_out_zoom[2,1]-BBox_out_zoom[2,0])/2.0-cutsize/2.0



density,den_k,den_fft,_ = initialize_density(ngrid,ngrid,ngrid)
density.fill(0.0)

# Lay down fine particles on density array with CiC:
CICDeposit_3(
                 px_zoom-com[0],
                 py_zoom-com[1],
                 pz_zoom-com[2],
                 px_zoom,py_zoom,pz_zoom, #dummies
                 density,

                 cellsize_zoom,
                 cutsize/float(ngrid),
                 0,
                 0,      # dummy
                 0,      # dummy

                 np.array([[0,0],[0,0],[0,0]],dtype='int32'),

                 np.array([0.0,0.0,0.0],dtype='float32'),
                 0)

# Lay down any present crude particles on density array with CiC:
CICDeposit_3(
                 px-com[0],
                 py-com[1],
                 pz-com[2],
                 px,py,pz, #dummies
                 density,

                 cellsize,
                 cutsize/float(ngrid),
                 0,
                 0,      # dummy
                 0,      # dummy

                 BBox_in,

                 np.array([0.0,0.0,0.0],dtype='float32'),
                 0)

# make the figure:
plt.imshow(np.arcsinh((density.mean(axis=2))*np.sinh(1.0)/10.0)**(1./3.),
           vmin=0.0,vmax=1.75,
           interpolation='bicubic',cmap='CMRmap_r')
plt.axis('off')
plt.show()
```

[MUSIC] *Multi-scale initial conditions for cosmological simulations*, O. Hahn, T. Abel, Monthly Notices of the Royal Astronomical Society, 415, 2101 (2011), arXiv:1103.6031. The code can be found on this website.

[spatialCOLA] *Extending the N-body Comoving Lagrangian Acceleration Method to the Spatial Domain*, S. Tassev, D. J. Eisenstein, B. D. Wandelt, M. Zaldarriaga, (2014), arXiv:14??.????

[temporalCOLA] *Solving Large Scale Structure in Ten Easy Steps with COLA*, S. Tassev, M. Zaldarriaga, D. J. Eisenstein, Journal of Cosmology and Astroparticle Physics, 06, 036 (2013), arXiv:1301.0322

INDEX