Listening to the Sound of the Early Universe

Qing Zhou, s2501597

May, 2020

Abstract

The accelerating expansion of the universe is one of the most intriguing discovery made in cosmology. To further investigate the properties of the drive of this expansion – dark energy, efforts have been made both observationally and simulationally, in order to constrain the cosmological parameters to test the cosmological models. Amongst which baryonic acoustic oscillations (BAO) is with the least systematics, complementary to the most probed technique – supernovae Ia. In this project, we aim to observe the BAO signal from a large N-body simulation, using the GADGET-2 code (Springel, 2005). Two runs were carried out, one with wiggles in the initial power spectrum and the other without (for comparison). We have observed the oscillatory feature of the BAO signal, and have also estimated the sound horizon to be $\sim 136\,\mathrm{Mpc}$, with an error of $\sim 9\%$.

1 Introduction

In cosmology, baryon acoustic oscillations (BAO) are fluctuations in the density field of the baryonic matter in the early universe. During the radiation epoch, photons and baryons are coupled to each other, leading to acoustic oscillations in the primordial photon-baryon plasma. This tight coupling between electrons, baryons and photons due to Compton scattering then causes baryons to oscillate in phase with radiation. Thus, the whole plasma oscillates due to the sound waves. More importantly, just as supernovae provide "standard candles" for astronomical observations, BAO also provides a "standard ruler" in cosmology. BAO measurements is a powerful tool helping cosmologists understand the nature of dark energy by constraining cosmological parameters.

2 The BAO Signal and Power Spectrum

Prior to redshifts around 1000, the universe is a hot dense plasma of ionised gas. The cross section of the free electrons is sufficient for the CMB photons to Thomson scatter, resulting in a mean free time less than the Hubble time. The net result is a close coupling between the electrons baryons and photons. The radiation pressure of the photons is larger than the gravitational forces between the baryons and dark matter, resulting in perturbations in the primordial photon-baryon fluid to oscillate like sound waves. Diffusion of photons relative to baryons damps these oscillations, a phenomenon known as "Silk damping". After recombination, the mean free time of the photons became longer than the Hubble time. Photons decouple from the baryons, and the perturbations in the baryons soon became smoothly distributed, just like the perturbations in cold dark matter.

2.1 Density field and correlation function

Since the BAO signal is imprint of the density perturbations in the primordial plasma, our story begins with the density field. Let us focus on a continuous density field over a finite cube volume $V = L^3$ with periodic boundary conditions applied to all three directions. A dimensionless overdensity $\delta(\boldsymbol{x})$ can be written in terms of matter density $\rho(\boldsymbol{x})$

$$\delta(\boldsymbol{x}) = \frac{\rho(\boldsymbol{x})}{\bar{\rho}} - 1 \tag{1}$$

where $\bar{\rho}$ is the mean density in the volume V. In Fourier space this becomes

$$\delta(\boldsymbol{k}) = \int_V \frac{d^3x}{(2\pi)^3} e^{-i\boldsymbol{k}\cdot\boldsymbol{x}} \delta(\boldsymbol{x}) \tag{2}$$

And the poower spectrum is related to the density by

$$P(k) \equiv \langle |\delta(\boldsymbol{k})|^2 \rangle \tag{3}$$

where $\langle \cdots \rangle$ represents the ensemble average

$$\langle \delta(\boldsymbol{k}_1)\delta(\boldsymbol{k}_2) \rangle = \frac{\delta^K_{\boldsymbol{k}_{12}}}{k_f^3} P(k_1) \tag{4}$$

with $\boldsymbol{k}_{i_1,\cdots,i_n} \equiv \boldsymbol{k}_{i_1} + \cdots \boldsymbol{k}_{i_n}$ and $\delta^K_{\boldsymbol{k}}$ being a Kronecker delta. For $V \to \infty$, $k_f \to 0$, and $\delta^K_{\boldsymbol{k}}/k_f^3 \to \delta_D(\boldsymbol{k})$ becomes a Dirac delta[1][2]. However, in practice we have an N-boody simulation with finite particle numbers N_p with positions $\{x_i\}$ for $i = 1, \cdots, N_p$. In this case the density becomes

$$\rho(\boldsymbol{x}) = \sum_i^{N_p} m\delta_D(\boldsymbol{x} - \boldsymbol{x}_i) \tag{5}$$

where m is the particle mass. It follows that

$$\delta(\boldsymbol{x}) = \frac{1}{\bar{n}} \sum_i^{N_p} \delta_D(\boldsymbol{x} - \boldsymbol{x}_i) - 1 \tag{6}$$

and its Fourier transform becomes

$$\delta(\boldsymbol{k}) = \frac{1}{(2\pi)^3} \frac{1}{\bar{n}} \sum_i^{N_p} e^{-i\boldsymbol{k}\cdot\boldsymbol{x}_i} - \frac{\delta^K_{\boldsymbol{k}}}{k_f^3} \tag{7}$$

and the estimator of the power spectrum is given by

$$\hat{P}(k) = k_f^3 |\delta(\boldsymbol{k})|^2 = \delta_f^3 \left[|\delta(\boldsymbol{k})|^2 - \frac{1}{N_p} \right] \tag{8}$$

2.2 Cloud-in-Cell mass assignment scheme

Getting the density field straight from the discrete particle distribution is slow. A more efficent way of obtaining the density field, is the so-called Cloud-in-Cell (CIC) mass assignment scheme. It takes advantage of the FFT algorithm, and requires first order interpolation of the density field on a regular grid in position space. The explicit form of the CIC window function reads

$$W_{CIC}(x) = \frac{1}{H} \begin{cases} 1 - |x|/H, |x| < H \\ 0, \qquad\quad \text{otherwise.} \end{cases} \tag{9}$$

with N_G the linear size of the grid and $H = L/N_G$ the grid spacing. The window function can be easily extended to three dimensions

$$W(\boldsymbol{x}) = W(x_1)W(x_2)W(x_3) \tag{10}$$

whose Fourier transform is[3]

$$W(\boldsymbol{k}) = \left[\mathrm{sinc}\left(\frac{\pi \mathrm{k}_1}{2\mathrm{k}_\mathrm{N}}\right)\left(\frac{\pi \mathrm{k}_2}{2\mathrm{k}_\mathrm{N}}\right)\left(\frac{\pi \mathrm{k}_3}{2\mathrm{k}_\mathrm{N}}\right) \right]^2 \tag{11}$$

where $k_N = \pi/H$ is the Nyquist frequency. While this method provides a. resonable and effective solution, it also introduces an alias in the power spectrum estimation. After the window function, the estimator of the power spectrum is now related to the correlation function $\langle |\delta(\boldsymbol{k})|^2 \rangle$ by

$$\langle |\delta(\boldsymbol{k})|^2 \rangle = \sum_i^{N_p} |W(\boldsymbol{k} + 2k_N\boldsymbol{x}_i)|^2 P(k + 2k_N\boldsymbol{x}_i) + \frac{1}{N} \sum_i^{N_p} |W(\boldsymbol{k} + 2k_N\boldsymbol{x}_i)|^2 \tag{12}$$

[1] $\delta_D(\boldsymbol{x}) = \frac{1}{V}\sum_{\boldsymbol{k}} e^{i\boldsymbol{k}\cdot\boldsymbol{x}}$

[2] $\delta^K_{\boldsymbol{k}} = \frac{1}{V}\int d^3x e^{-i\boldsymbol{k}\cdot\boldsymbol{x}}$

where the summation is over all 3D integer vectors $\boldsymbol{x}_i$. The density convolution introduces a factor W^2 both to the power spectrum and the shot noise (the $1/N$ term). The analytical form of the shot noise can be expressed as

$$D^2(\boldsymbol{k}) = \frac{1}{N}\sum_{i}^{N_p} |W(\boldsymbol{k} + 2k_N \boldsymbol{x}_i)|^2 = \frac{1}{N} C_1(\boldsymbol{k}) \tag{13}$$

with

$$C_1(\boldsymbol{k}) = \Pi_i \left[1 - \frac{2}{3}\sin^2\left(\frac{\pi k_i}{2k_N}\right)\right] \tag{14}$$

where $\Pi_i(\boldsymbol{k})$ is the Fourier transform of the sampling function $\Pi(\boldsymbol{x})$

$$\Pi(\boldsymbol{x}) = \sum_{i}^{N_p} \delta_D(\boldsymbol{x} - \boldsymbol{x}_i) \tag{15}$$

Therefore after correcting the shot noise effect we can recover the true power spectrum from (12).

Finally, since the BAO signal is weak, to subtract it from the power spectrum, we divide it by a smoothed power spectrum in order to see the oscillations produced by the baryons

$$\mathcal{S}_{\mathrm{BAO}} = \frac{P(k)}{P^{\mathrm{smoothed}}(k)} - 1 \tag{16}$$

3 N-body Simulation

In this project, we ran a large N-body simulation with a box size of 500 Mpc/h and 512^3 dark matter particles. The particles were evolved from $z_{ini} = 127$, up to redshift zero, using the $Gadget-2$ code[4]. Our simulation adopt cosmological parameters from WMAP9[5]: $\Omega_M = 0.279$, $\Omega_\Lambda = 0.721$, $\Omega_b = 0.0463$, and $H_0 = 70\,\mathrm{km\,s^{-1}\,Mpc^{-1}}$. We have in total two runs, one whose initial power spectrum is with wiggles and the other without (used only for comparison in recovering the BAO signal). For each run we have stored 13 snapshots. Four snapshots of the cubic volume of particles projected onto the xy-plane is shown in Fig. 1. It can be seen, initially at high redshifts (panel 1 and 2), the projection of the density field is somewhat uniform. In lower redshifts (panel 3 and 4), large scale structures represented by the darker regions in the figures can be seen delineating voids which are the underdensities.

After mass assignment using the CIC scheme, the density field can be seen as in Fig. 2.

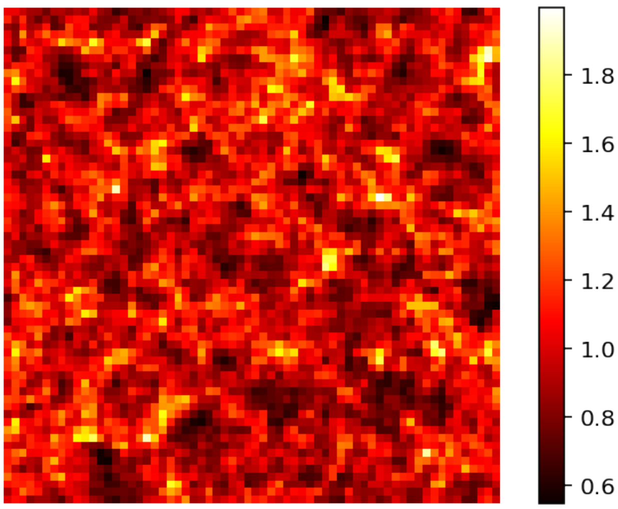

Figure 2: The projected image of the density field at redshift zero, after the mass assignment scheme. The grid number used here is 64^3, colour bar represents the density, where hotter regions have higher values in colour.

Afterwards we can recover the power spectrum from the Fourier transform of the correlation function, which can be seen in Fig. 3

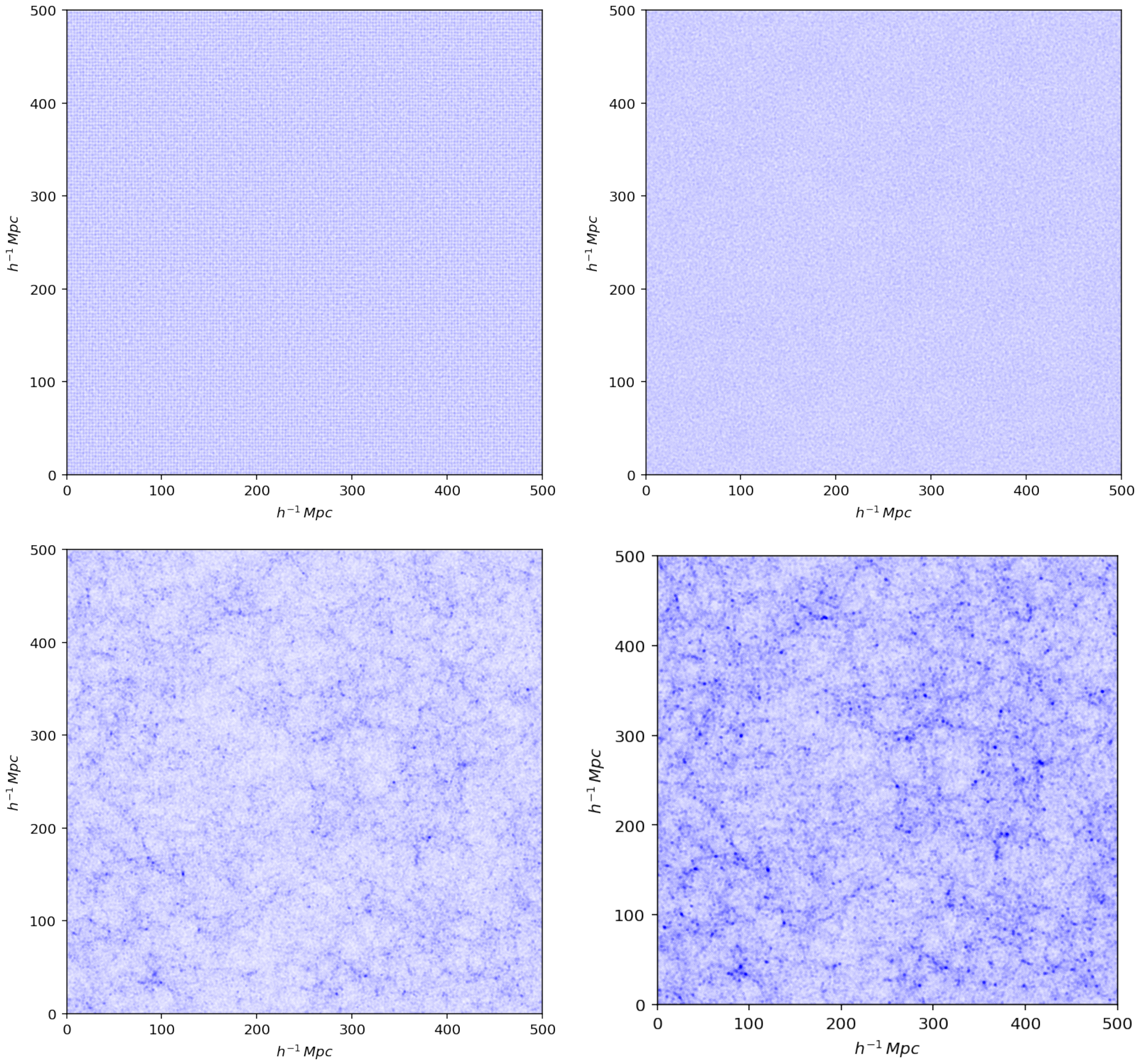

Figure 1: The projected image of the density field at various redshifts (z=127, 10, 0.4, 0), for 512^3 dark matter particles in a cubic volume of $(500\,\mathrm{h}^{-1}\,\mathrm{Mpc})^3$. Filament-like structures can be seen delineating voids at redshift zero.

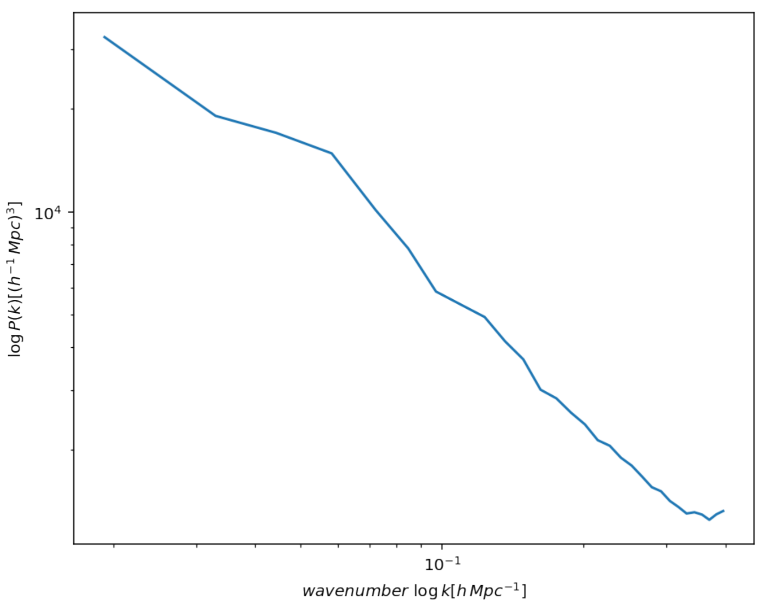

Figure 3: The power spectrum at redshift zero, recovered using the CIC scheme with a grid number of 64.

In order to see at what resolution of grid size the power spectrum can be deemed converged, we generate multiple spectra using different grid number. Fig. 4 shows the resultant power spectra.

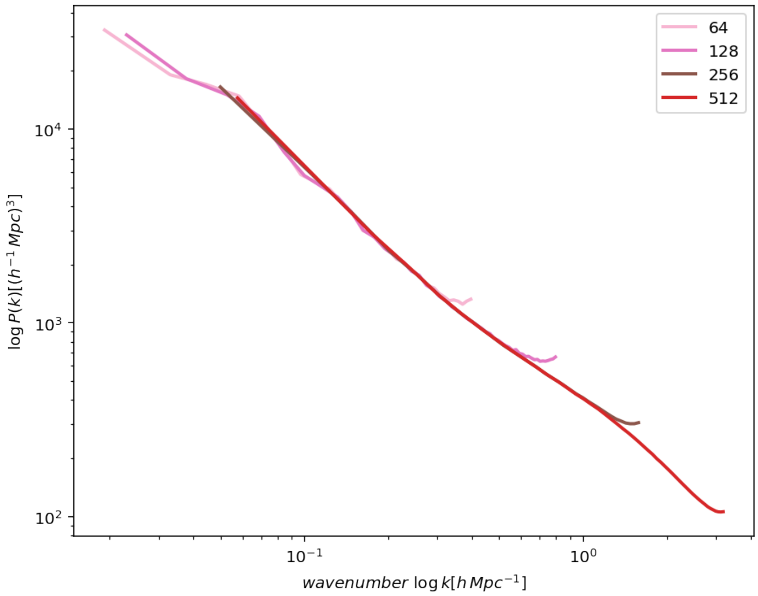

Figure 4: The power spectra at redshift zero, with varying grid sizes. The grid number used are 64^3, 128^3,256^3, and 512^3.

Undoubtedly the spectrum with 512^3 grids has the least noise, albeit this is computationally demanding. The grid size is already equal to the particle number. Comparing it to the spectrum with 256^3 grids, we find the latter well converged within range $[10^{-1}, 10^0]\log k$. Hence for recovering the BAO signal, it is sufficient to use the spectrum with 256^3 grids.

Now using 256^3 grids we can obtain the power spectra at various redshifts (z=127, 10, 0.4, 0)See Fig. 5. The lower the redshift, the higher the amplitude the power spectrum would be. This is due to the fact that at lower redshifts the clustering is stronger, which can be seen when compared to Fig. 1. With the decrease of redshifts, more structures would form. The gaps between the power spectra at larger scales (correspond to smaller value in wavenumber k) evolves linearly according to perturbation theory, however this is not true for the evolution of the small scale structures. To see this more clearly we can subtract the linear effect.

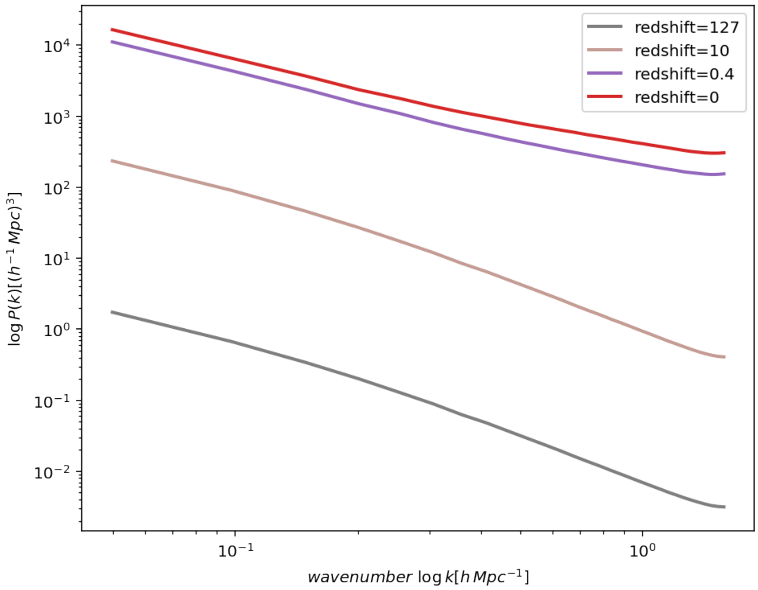

Figure 5: The power spectra at various redshifts (z=127, 10, 0.4, 0).

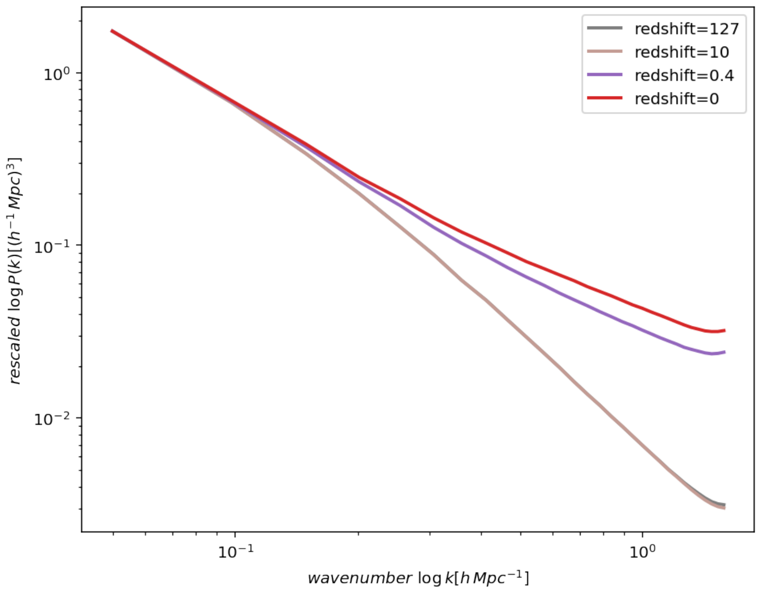

Figure 6: Same as 5, but now rescaled to the amplitude of the spectrum of $z = 0$.

If we rescale the power spectra so that they all begin with the same amplitude of the $z = 0$'s, we can see the effect of non-linear evolution in the small scale more clearly. From Fig. 6 it can be seen, the clustering at smaller scales is stronger at a later time epoch. This is because galaxy clusters form in a latter epoch in the evolution of the universe.

The BAO signal is finally recovered by dividing the power spectra by their smoothed counterparts. The evolution of the BAO signal can be seen in Fig. 8. To understand how the features form we first refer to Fig. 7[6]. In the beginning, the density field is smooth, the fluctuations are the same on all directions. As time goes by, this symmetry breaks down at smaller scales, particles originally located at the sound horizon gets "perturbed" around the ring, broadening the acoustic feature.

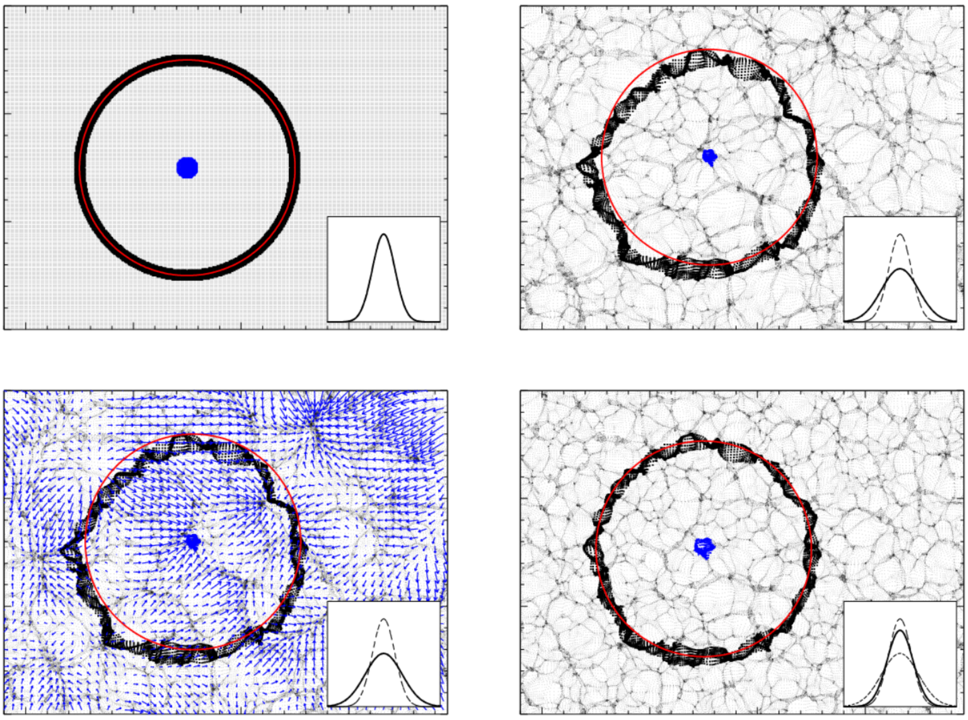

Figure 7: In each panel, we show a thin slice of a simulated cosmological density field. (top left) In the early universe, the initial densities are very smooth. We mark the acoustic feature with a ring of 150 Mpc radius from the central points. A Gaussian with the same rms width as the radial distribution of the black points from the centroid of the blue points is shown in the inset. (top right) We evolve the particles to the present day, here by the Zel'dovich approximation (Zel'dovich 1970). The red circle shows the initial radius of the ring, centered on the current centroid of the blue points. The large-scale velocity field has caused the black points to spread out; this causes the acoustic feature to be broader. The inset shows the current rms radius of the black points relative to the centroid of the blue points (solid line) compared to the initial rms (dashed line). (bottom left) As before, but overplotted with the Lagrangian displacement field, smoothed by a $10\mathrm{h}^{-1}$ Mpc Gaussian filter. The concept of reconstruction is to estimate this displacement field from the final density field and then move the particles back to their initial positions. (bottom right) We displace the present-day position of the particles by the opposite of the displacement field in the previous panel. Because of the smoothing of the displacement field, the result is not uniform. However, the acoustic ring has been moved substantially closer to the red circle. The inset shows that the new rms radius of the black points (solid), compared to the initial width (long-dashed) and the uncorrected present-day width (short-dashed). The narrower peak will make it easier to measure the acoustic scale[6].

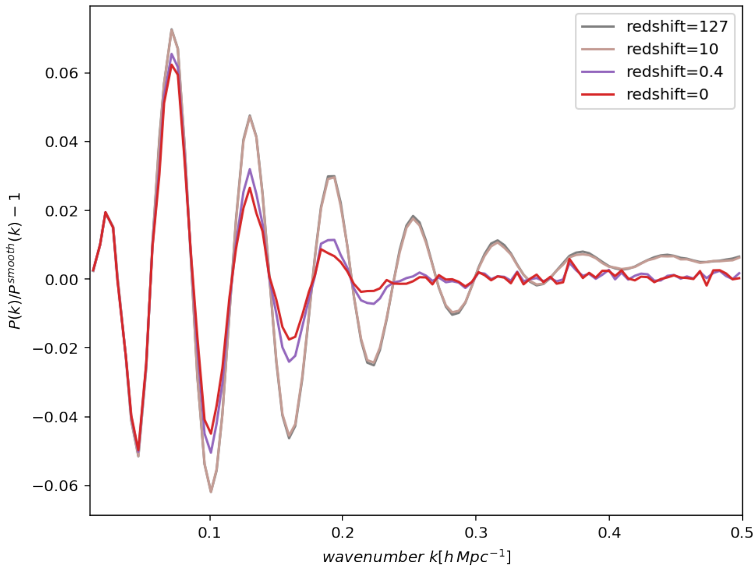

Figure 8: The BAO signal at various redshifts (z=127, 10, 0.4, 0).

Overally, oscillating wave structures can be seen in the BAO signal. As time goes by, the damping features in the tail are gradually smoothed. The BAO signal can be understood as a standing wave in Fourier space. In Fourier space, a single acoustic scale can result in a harmonic sequence of oscillations in the power spectrum, imprinting the response of the universe to a plane wave perturbation. Each wave crest in the initial wave produces a planar sound wave travelling a distance equal to the acoustic scale. If the sound wave is on the peak of the dark matter perturbation, then it gets constructive interference; otherwise it gets destructive interference. Hence there is a harmonic relation between the perturbation wavelength and the acoustic scale. The smearing-out of the features on the tail is due to the broadening in the acoustic waves, caused by the large-scale velocity field. Moreover, we can use the scale of the maximum peak to estimate the sound horizon. From Fig. 8, the scale of the maximum peak is at $k = 0.0657\,\mathrm{h\,Mpc}^{-1}$, corresponds to a horizon size of $\sim 136\,\mathrm{Mpc}$. However, due to the non-linear gravitational evolution of the acoustic wave, the scales are shifted and waves broadened, resulting in a 9.3% deviation from the value determined from other surveys. On the other hand, the density distribution is not uniform, pairs of overndensities fall toward each other, and pairs of underdensities fall away from each other, contributing to the systematics in the two point correlation,causing a partial cancellation.

4 Conclusions

In this project, we have carried out an N-body simulation in order to recover the power spectrum, and recover the initial BAO signal. We have observed the initial oscillating waves as a result of the density perturbations in the primordial plasma of coupled baryons and photons. An observed shift and broadening in the acoustic scale is noticed, caused by the non-linear structure formation in the low redshift universe. Using the scales of the acoustic peaks we can further constrain cosmological parameters from the inferred sound horizon. To further improve the precision, it is useful to apply a technique known as ”reconstruction”.

References

[1] Aubourg, E., Bailey, S., Bautista, J. E., et al. 2015, PRD, 92, 123516

[2] Jing, Y. P. 2005, ApJ, 620, 559

[3] Hockney, R.W. & Eastwood, J.W. 1981, Computer simulations using particles. Mc Graw-Hill

[4] Springel V., 2005, MNRAS, 364, 1105

[5] Hinshaw, G., Spergel, D. N., Verde, L., et al. 2003, ApJS, 148, 135

[6] Padmanabhan, N., Xu, X., Eisenstein, D. J., et al. 2012, MNRAS, 427, 2132

5 Appendix

```python
#!/usr/bin/env python
# coding: utf-8

import numpy as np
import os
import logging
import matplotlib.pyplot as plt
logging.basicConfig(level=logging.INFO)
import matplotlib as mpl


from cic import cic
from read_snapshot import ReadSnapshot
from ps import PowerSpectrum


OutBase = "/data/dell5/userdir/test/Qing"


Ng = 64
L = 500
binsnum = 30


snap = 12
Base =
    '/data/dell5/userdir/BAO/par_dependence/SIMs/{}_WMAP9/output/snapdir_{:03d}/snapshot_{:03d}.'


Base.format('nw01', snap, snap)


rs = ReadSnapshot(Base.format('w01', snap, snap))



def func_readpos(name='nw01', snap=snap):
    rs = ReadSnapshot(Base.format(name, snap, snap))
    logging.info("redshift: %f"%rs.Info['redshift'][0])
    Pos = []
    for i in np.arange(rs.Info['Nsubfiles'][0]):
        pos = rs.ReadPos(Filenum=i)
        Pos.append(pos['pos'])
    Pos = np.vstack(Pos)
    return Pos

def func_cic(Pos, Ng, L, name, snap=snap):
    grid = cic(Pos, Ng=Ng, L=L)
    logging.info('Check: {} should be {}!'.format(
        grid.astype(np.float64).sum(), rs.Info['npartall'][0, 1]))
    grid /= grid.mean()
    np.save(os.path.join(OutBase, '{}_cic_Ng{}_snap{}.npy'.format(name, Ng, snap)), grid)
    return grid


def func_ps(grid, Ng, L, binsnum, name, snap=snap, BAO=False):
    ps = PowerSpectrum(L=float(L), Ng=Ng)
    ps.set_binsnum(binsnum)
    if BAO:
        ps.set_binskrange(1e-2, 0.5)
    ps.set_binstyle('linear')
    ps.AutoPS(grid)
    k, pk, nbins = ps.get_bin1d()
    if BAO:
        path = '{}_BAO_Ng{}_snap{}.txt'.format(name, Ng, snap)
    else:
        path = '{}_ps_Ng{}_snap{}.txt'.format(name, Ng, snap)
    np.savetxt(os.path.join(OutBase, path), np.c_[k, pk, nbins])
    return k, pk, nbins
```

```python



def pipeline(name, snap, Ng, L, binsnum, BAO=False):
    pos = func_readpos(name, snap)
    grid = func_cic(pos, Ng, L, name, snap)
    ps = func_ps(grid, Ng, L, binsnum, name, snap, BAO=BAO)


Pos = func_readpos('w01', 12)



plt.figure(figsize=(5,5))
plt.plot(Pos[::100, 0], Pos[::100, 1], 'b.', alpha=0.1, ms=0.1)
plt.xlim([0,500])
plt.ylim([0,500])
plt.xlabel(r'$h^{-1}\,Mpc$')
plt.ylabel(r'$h^{-1}\,Mpc$')
plt.show()


grid = func_cic(Pos, Ng, L, name='w01')


plt.imshow(grid.mean(0),cmap='hot')
plt.axis('off')
plt.colorbar()
plt.show()



plt.imshow(grid.mean(1),cmap='hot')
plt.axis('off')
plt.colorbar()
plt.show()


k, pk, nbins = func_ps(grid, Ng, L, binsnum, 'w01')



plt.figure(figsize=(7.5,6))
plt.loglog(k, pk)
plt.xlabel(r'$wavenumber \ \log \,k[h\,Mpc^{-1}]$')
plt.ylabel(r'$\log \,P(k)[(h^{-1}\,Mpc)^3]$')
plt.show()

for i in [64,128,256,512]:
    pipeline('w01', snap=12, Ng=i, L=L, binsnum=binsnum)

import matplotlib.pylab as pl

def showps(Ng):
    colors = pl.cm.tab20(np.linspace(0.3,0.7,512))[::-1]
    color = colors[Ng-1]
    pk = np.loadtxt(os.path.join(OutBase, '{}_ps_Ng{}_snap{}.txt'.format('w01', Ng, 12)))
    plt.plot(pk[:,0], pk[:,1], lw=2,color=color, label='{}'.format(Ng))



plt.figure(figsize=(7.5,6))
for i in [64, 128, 256, 512]:
    showps(i)
plt.legend()
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$wavenumber \ \log \,k[h\,Mpc^{-1}]$')
plt.ylabel(r'$\log \,P(k)[(h^{-1}\,Mpc)^3]$')
plt.show()


```

```
for i in [0, 4, 8, 12]:
    pipeline('w01', snap=i, Ng=256, L=L, binsnum=30, BAO=False)


snap = 0
z = [127,10,0.4,0]
colors = pl.cm.tab20(np.linspace(0.3,0.7,13))[::-1]
plt.figure(figsize=(7.5,6))
for j,snap in enumerate([0, 4, 8, 12]):
    color = colors[snap]
    pkw = np.loadtxt(os.path.join(OutBase, '{}_ps_Ng{}_snap{}.txt'.format('w01', 256, snap)))
    plt.plot(pkw[:,0], pkw[:,1], lw=2, color=color, label='redshift={}'.format(z[j]))
plt.legend()
plt.xscale('log')
plt.yscale('log')
plt.xlabel(r'$wavenumber \ \log \,k[h\,Mpc^{-1}]$')
plt.ylabel(r'$\log \,P(k)[(h^{-1}\,Mpc)^3]$')
plt.show()




for i in [0, 4, 8, 12]:
    pipeline('w01', snap=i, Ng=256, L=L, binsnum=100, BAO=True)
    pipeline('nw01', snap=i, Ng=256, L=L, binsnum=100, BAO=True)




snap = 0
plt.figure(figsize=(7.5,6))
for j,snap in enumerate([0, 4, 8, 12]):
    color = colors[snap]
    pkw = np.loadtxt(os.path.join(OutBase, '{}_BAO_Ng{}_snap{}.txt'.format('w01', 256, snap)))
    pknw = np.loadtxt(os.path.join(OutBase, '{}_BAO_Ng{}_snap{}.txt'.format('nw01', 256,
    snap)))
    S = (pkw[:,1] - pknw[:,1])/pknw[:,1]
    plt.plot(pkw[:,0], S, color=color,label='redshift={}'.format(z[j]))
    ind = np.argmax(S[8:12])
    print(pkw[:,0][8:12][ind])
plt.xlim([1e-2, 0.5])
plt.xlabel(r'$wavenumber \  k[h\,Mpc^{-1}]$')
plt.ylabel(r'$P(k)/P^{smooth}(k)-1$')
plt.legend()
plt.show()
```

main.py

```
#!/usr/bin/env python
# coding=utf-8
import logging
import numpy as np

log = logging.info


class PowerSpectrum(object):
    def __init__(self, L=300., Ng=128, **kwargs):
        self.L = L
        self.Ng = Ng
        log('=' * 80)
        log('parameters:')
        log('N: %d, L: %f' % (self.Ng, self.L))
        log('=' * 80)

        self.Kf = 2 * np.pi / self.L
        self.H = self.L / self.Ng
        self.fn = np.fft.fftfreq(self.Ng, 1. / self.Ng)
        # fn: 0,1,2,...,511,-512,-511,-510,...,-2,-1
        self.k_ind = (self.fn[:, None, None]**2.
                      + self.fn[None, :, None]**2.
```

```
                      + self.fn[None, None, :]**2)**(0.5)

    def set_binsnum(self, num):
        self.bins = num

    def set_binstyle(self, style=''):
        if style == 'log':
            self.binspace = self.logspace_wrapper(np.logspace)
        elif style == 'linear':
            self.binspace = np.linspace
        else:
            raise ValueError('binstyle should be "log" or "linear".')

    def set_binskrange(self, kmin, kmax):
        """ input wave number, not grid index"""
        self.bins_kmin = kmin
        self.bins_kmax = kmax

    def logspace_wrapper(self, func):
        def wrapper(*args, **kwargs):
            a = args[0]
            b = args[1]
            args = args[2:]
            return func(np.log10(a), np.log10(b), *args, **kwargs)
        return wrapper

    def fft(self, data):
        assert data.ndim == 3
        datak = np.fft.fftn(data, norm=None)
        return datak

    def _DeWindow(self):
        window = (np.sinc(1. / self.Ng * self.fn[:, None, None])
                  * np.sinc(1. / self.Ng * self.fn[None, :, None])
                  * np.sinc(1. / self.Ng * self.fn[None, None, :]))
        return window

    def AutoPS(self, delta, dew=True):
        deltak = self.fft(delta)
        if dew:
            window = self._DeWindow()
            deltak /= window ** 2  # CIC deconvolve window function
        self.pk = np.abs(deltak)**2.

    def get_bin1d(self):
        try:
            max = self.bins_kmax / self.Kf
            min = self.bins_kmin / self.Kf
        except AttributeError:
            max = self.Ng/2
            min = 1
        try:
            bin = self.binspace(min, max, self.bins+1, endpoint=True)
        except AttributeError:
            log('can not find parameter binstyle, \
                        use default logspace...')
            bin = np.logspace(np.log10(min), np.log10(max),
                              self.bins+1, endpoint=True)
        n_bin = []
        Pk_bin = []
        k_bin = []
        for i in range(self.bins):
            bool = (bin[i] <= self.k_ind) * (self.k_ind < bin[i+1])
            n_bin.append(bool.sum())
            k_bin.append(self.k_ind[bool].astype(np.float64).mean())
            Pk_bin.append(self.pk[bool].astype(np.float64).mean())
        n_bin = np.array(n_bin)
        k_bin = np.array(k_bin)
        Pk_bin = np.array(Pk_bin)
        n_bin[n_bin == 0] = 1.
        k_bin *= self.Kf
        Pk_bin *= self.L**3/self.Ng**6
```

```python
        return k_bin, Pk_bin, n_bin
```

ps.py

```python
#!/usr/bin/env python
# coding=utf-8
import numpy as np
import logging

logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.INFO)


class ReadSnapshot():
    'read data for L-Gadget'

    def __init__(self, Path=''):
        '''
        usage:
        Path='/data/dell1/userdir/output/gadget/test/snapdir_005/snapshot_005.'
        NumOfFile=64
        '''
        self.Path = Path
        self.dt = np.dtype([('head', np.int32, 1),
                            ('npart', np.int32, 6),
                            ('massarr', np.float64, 6),
                            ('time', np.float64),
                            ('redshift', np.float64),
                            ('flag_sfr', np.int32),
                            ('flag_feedback', np.int32),
                            ('npartall', np.int32, 6),
                            ('flag_cooling', np.int32),
                            ('Nsubfiles', np.int32),
                            ('BoxSize', np.float64),
                            ('Omega0', np.float64),
                            ('OmegaL', np.float64),
                            ('H', np.float64),
                            ])
        f = open(self.Path + '0', 'r')
        self.Info = np.fromfile(file=f, dtype=self.dt, count=1)
        f.close()
        self.Filenum = self.Info['Nsubfiles'][0]
#       print 'Path:', Path
#       print self.Info

    def ReadPos(self, Filenum=np.int):
        dtype_data = np.dtype([('pos', np.float32, 3)])
        f = open(self.Path + '%d' % Filenum, 'r')
        info = np.fromfile(file=f, dtype=self.dt, count=1)
        length = info['npart'][0][1]
#       print 'reading pos:'
        f.seek(self.Info['head'][0] + 4 + 4)
        a = np.fromfile(file=f, dtype=np.int32, count=1)[0]
        pos = np.fromfile(file=f, dtype=dtype_data, count=info['npart'][0][1])
        b = np.fromfile(file=f, dtype=np.int32, count=1)[0]
        f.close()
        if not (a == b and a == (length * 4 * 3)):
            logging.info('error in pos!')
            logging.info(a)
            logging.info(b)
        else:
            return pos

    def ReadVel(self, Filenum=np.int):
        dtype_data = np.dtype([('vel', np.float32, 3)])
        f = open(self.Path + '%d' % Filenum, 'r')
        info = np.fromfile(file=f, dtype=self.dt, count=1)
        length = info['npart'][0][1]
        logging.info('reading vel:')
        f.seek(self.Info['head'][0] + 4 + 4 + length * 4 * 3 + 4 + 4)
        a = np.fromfile(file=f, dtype=np.int32, count=1)[0]
        vel = np.fromfile(file=f, dtype=dtype_data, count=info['npart'][0][1])
        b = np.fromfile(file=f, dtype=np.int32, count=1)[0]
```

```
        f.close()
        if not (a == b and a == (length * 4 * 3)):
            logging.info('error in pos!')
            logging.info(a)
            logging.info(b)
        else:
            return vel

    def ReadPID(self, Filenum=np.int):
        f = open(self.Path + '%d' % Filenum, 'r')
        info = np.fromfile(file=f, dtype=self.dt, count=1)
        length = info['npart'][0][1]
        logging.info('reading PID:')
        f.seek(
            self.Info['head'][0] + 4 + 4 +
            length * 4 * 3 + 4 + 4 +
            length * 4 * 3 + 4 + 4
        )
        a = np.fromfile(file=f, dtype=np.int32, count=1)
        PID = np.fromfile(file=f, dtype=np.int64, count=info['npart'][0][1])
        b = np.fromfile(file=f, dtype=np.int32, count=1)
        f.close()
        if not (a == b and a == (length * 4 * 2)):
            logging.info('error in PID!')
            logging.info(a)
            logging.info(b)
        else:
            return PID


if __name__ == '__main__':
    import sys
    Path = sys.argv[1]
    NumOfFile = 64
    f = ReadSnapshot(Path)
    logging.info(f.Info)
    logging.info(f.Info.dtype)
```

read_snapshot.py

```
#!/usr/bin/env python
# coding=utf-8
import numpy as np
import matplotlib.pyplot as plt
import hmf
from astropy.cosmology import FlatLambdaCDM



name='WMAP9'

ips_camb  = np.loadtxt('WMAP9_matterpower.dat')

# no-wiggle
from nowigglePS import TF
k = ips_camb[:,0]
pk = ips_camb[:,1]
TFnw = TF(my_k = k,
    sigma_8=0.821, n=0.972, z=0,
    lnk_min=np.log(1e-3), lnk_max=np.log(1e3), dlnk=0.05,
    transfer_model=hmf.transfer_models.EH_NoBAO,
    cosmo_model=hmf.cosmo.WMAP9,
    )
pk = TFnw.SmoothWiggle(pk, la=0.25)

pk *= k**3./(2*np.pi**2.)
ips_nw = np.c_[k,pk]
bool = (ips_nw[:,0]>1e-3)*(ips_nw[:,0]<80)
ips_nw = ips_nw[bool]
ips_nw = np.log10(ips_nw)
np.savetxt('./inips/nowiggle_%s.dat'%(name), ips_nw)
# no-wiggle
```

```python


ips_camb[:,1] *= ips_camb[:,0]**3./(2*np.pi**2.)
ips_camb = np.log10(ips_camb)
ips_camb = ips_camb[bool]
np.savetxt('./inips/wiggle_%s.dat'%(name), ips_camb)
if __name__ == "__main__":
    plt.figure('ps')
    plt.plot(ips_camb[:,0], ips_camb[:,1], 'r-')
    plt.plot(ips_nw[:,0], ips_nw[:,1], 'b--', label='nw')
    plt.legend()
    plt.figure('BAO')
    plt.plot(ips_nw[:,0], 10**ips_camb[:,1]/10**ips_nw[:,1])
    plt.show()
```

create_PS–GadgetIC_WMAP.py

```python
#!/usr/bin/env python
# coding=utf-8
import hmf
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate

class TF(hmf.transfer.Transfer):
    def __init__(self, my_k, **kwargs):
        super(TF, self).__init__(**kwargs)
        self.my_k = my_k
        self.PS_EH()
        self.PS_EHnw()
        self.PS_BBKS()

    def interpolate(self, x_new):
        tck = interpolate.splrep(self.inpo_x, self.inpo_y)
        y_bspline = interpolate.splev(x_new, tck)
        return y_bspline

    def PS_EH(self):
        self.transfer_model = hmf.transfer_models.EH
        self.PkEH = self.power.copy()
        self.inpo_x = np.log(self.k)
        self.inpo_y = np.log(self.PkEH)
        self.my_EH = np.exp(self.interpolate(np.log(self.my_k)))

    def PS_EHnw(self):
        self.transfer_model = hmf.transfer_models.EH_NoBAO
        self.PkEHnw = self.power.copy()
        self.inpo_x = np.log(self.k)
        self.inpo_y = np.log(self.PkEHnw)
        self.my_EHnw = np.exp(self.interpolate(np.log(self.my_k)))

    def PS_BBKS(self):
        self.transfer_model = hmf.transfer_models.BBKS
        self.PkBBKS = self.power.copy()
        self.inpo_x = np.log(self.k)
        self.inpo_y = np.log(self.PkBBKS)
        self.my_BBKS = np.exp(self.interpolate(np.log(self.my_k)))

    def SmoothWiggle(self, pk, la=0.25):
        Pk_approx = self.my_EHnw
        klog = np.log10(self.my_k)
        pk = pk/Pk_approx
        factor = 1./np.sqrt(2*np.pi)/la

        dqlog = np.zeros_like(klog)
        dqlog[1:-1] = (klog[2:] - klog[:-2])/2.
        dqlog[0] = klog[1] - klog[0]
        dqlog[-1] = klog[-1] - klog[-2]
        #convolve...
        pnw = np.zeros_like(pk)
        norm = np.zeros_like(pk)
        for i in range(len(pnw)):
```

```
                pnw[i] += factor*(dqlog*pk*np.exp(-0.5/la**2.*(klog-klog[i])**2.)).sum()
                norm[i] += factor*(dqlog*np.exp(-0.5/la**2.*(klog-klog[i])**2.)).sum()
            pnw /= norm
            pnw = pnw*Pk_approx
            return pnw

        def testNW(self):
            my_pknw = self.SmoothWiggle(self.my_EH)
            plt.plot(self.k, self.PkEH/self.PkEHnw, 'g-', label='EH/EHnw')
            plt.plot(self.my_k, self.my_EH/my_pknw, 'b-', label='EH/Mynw')
            plt.hlines(1., self.k[0], self.k[-1], colors='k', linestyles='dashed')
            plt.xscale('log')
            plt.xlim([0.03,0.6])
            plt.ylim([0.85,1.15])
            plt.legend()
            plt.show()

if __name__ == '__main__':
    my_k = np.linspace(0.01,1,200)
    params = {
        'lnk_min': np.log(1e-3),
        'lnk_max': np.log(10),
        'dlnk':0.01,
        'sigma_8':0.8344,
        'n':0.9624,
        'z':0.0,
        'transfer_model':hmf.transfer_models.EH_NoBAO,
    }
    #-----------------------------------------------
    tf=TF(my_k=my_k, **params)
    hmf.transfer.Transfer.parameter_info()
    exit()
    #tf.SmoothWiggle(tf.PkEH)
    tf.testNW()
```

nowigglePS.py

```
#!/usr/bin/env python
# coding=utf-8
import numpy as np
from libgrid._cic import lib, ffi


def cic(pos, Ng=32, L=300):
    posx = pos[:, 0]
    posy = pos[:, 1]
    posz = pos[:, 2]
    if not posx.flags['C_CONTIGUOUS']:
        posx = np.ascontiguousarray(posx, dtype=pos.dtype)
    if not posy.flags['C_CONTIGUOUS']:
        posy = np.ascontiguousarray(posy, dtype=pos.dtype)
    if not posz.flags['C_CONTIGUOUS']:
        posz = np.ascontiguousarray(posz, dtype=pos.dtype)
    pinx = ffi.cast("float *", posx.ctypes.data)
    piny = ffi.cast("float *", posy.ctypes.data)
    pinz = ffi.cast("float *", posz.ctypes.data)
    out = np.zeros([Ng**3], dtype=np.float32)
    pout = ffi.cast("float *", out.ctypes.data)
    lib.CIC(Ng, L, len(pos), pinx, piny, pinz, pout)
    return out.reshape(Ng, Ng, Ng)


if __name__ == "__main__":
    import unittest
    import numpy as np

    class testcic():

        def __init__(self):
            self.Ng = 4
            self.Boxsize = 4
            pos = [
```

```
            [1., 2., 3.], [0.3, 0., 0.], [3.9, 0.2, 2.6], [0., 0., 0.],
            [0., 1., 0.], [4.0, 4., 4.], [2.6, 3.9, 2.2], [1.3, 2.5, 0.7],
        ]
        self.pos = np.array(pos, dtype=np.float32)

    def weight_CIC(self, s):
        w = np.zeros_like(s)
        bool = np.abs(s) < 1.0
        w[bool] += 1 - np.abs(s[bool])
        return w

    def GetWeight(self, gx, gy, gz, pos, window_func):
        wx = window_func(gx - pos[0])
        wy = window_func(gy - pos[1])
        wz = window_func(gz - pos[2])
        w = wx * wy * wz
        return w

    def CIC(self):
        zeros = np.zeros(
            [self.Ng + 2, self.Ng + 2, self.Ng + 2], dtype=np.float32)
        gpos = np.arange(-1, self.Ng + 1) + 0.5
        gx = gpos[:, None, None] + zeros
        gy = gpos[None, :, None] + zeros
        gz = gpos[None, None, :] + zeros
        for i in range(self.pos.shape[0]):
            weight = self.GetWeight(
                gx, gy, gz, self.pos[i], self.weight_CIC)
            zeros += weight
        zeros[1, :, :] += zeros[-1, :, :]
        zeros[-2, :, :] += zeros[0, :, :]
        zeros[:, 1, :] += zeros[:, -1, :]
        zeros[:, -2, :] += zeros[:, 0, :]
        zeros[:, :, 1] += zeros[:, :, -1]
        zeros[:, :, -2] += zeros[:, :, 0]
        grid = zeros[1:-1, 1:-1, 1:-1]
        assert grid.sum() == self.pos.shape[0]
        assert grid.shape == (self.Ng, self.Ng, self.Ng)
        return grid

    def testGrid_CIC_CA(self):
        from CosmAna import CIC as Assign
        self.pos[self.pos == self.Boxsize] -= self.Boxsize
        grid = Assign(self.pos,
                      NG=self.Ng,
                      L=self.Boxsize)
        grid = grid.reshape(self.Ng, self.Ng, self.Ng)
        assert grid.sum() == self.pos.shape[0]
        grid_true = self.CIC()
        assert np.allclose(grid, grid_true)

    def testGrid_CIC_ffi(self):
        self.pos[self.pos == self.Boxsize] -= self.Boxsize
        grid = cic(self.pos,
                   Ng=self.Ng,
                   L=self.Boxsize)
        print('+'*80)
        print('test:')
        print(grid)
        assert grid.sum() == self.pos.shape[0]
        grid_true = self.CIC()
        print(grid_true)
        assert np.allclose(grid, grid_true)

t = testcic()
print(t.pos)
t.testGrid_CIC_ffi()
t.testGrid_CIC_CA()
```

cic.py

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>

int Index(int N, int x, int y, int z);

int CIC(int N, int L, int SIZE, float* posx, float* posy, float* posz, float* pout)
{
    int n;
    int ibin, jbin, kbin, ibinm, ibinp, jbinm, jbinp, kbinm, kbinp;
    float hx, hy, hz, hx0, hy0, hz0, hxp, hxm, hyp, hym, hzp, hzm;
    float H = (float)L / (float)N;
    double value = 1.;
    double *gridP = (double *)malloc(N * N * N * sizeof(double));
    memset(gridP, 0, sizeof(double) * N * N * N);  // initialization

    for (n = 0; n < SIZE; n++)
    {
        ibin = posx[n] / H;
        jbin = posy[n] / H;
        kbin = posz[n] / H;

        hx = posx[n] / H - ibin - 0.5;
        hy = posy[n] / H - jbin - 0.5;
        hz = posz[n] / H - kbin - 0.5;

        hx0 = 1. - fabs(hx);
        hy0 = 1. - fabs(hy);
        hz0 = 1. - fabs(hz);

        if (hx > 0)
        {
            hxp = hx;
            hxm = 0.;
        }
        else
        {
            hxp = 0.;
            hxm = -hx;
        }

        if (hy > 0)
        {
            hyp = hy;
            hym = 0.;
        }
        else
        {
            hyp = 0.;
            hym = -hy;
        }

        if (hz > 0)
        {
            hzp = hz;
            hzm = 0.;
        }
        else
        {
            hzp = 0.;
            hzm = -hz;
        }

        ibinm = (ibin - 1 + N) % N;
        ibinp = (ibin + 1 + N) % N;
        jbinm = (jbin - 1 + N) % N;
        jbinp = (jbin + 1 + N) % N;
        kbinm = (kbin - 1 + N) % N;
        kbinp = (kbin + 1 + N) % N;

        gridP[Index(N, ibinm, jbinm, kbinm)] += hxm * hym * hzm * value;
```

```c
        gridP[Index(N, ibinm, jbinm, kbin)] += hxm * hym * hz0 * value;
        gridP[Index(N, ibinm, jbinm, kbinp)] += hxm * hym * hzp * value;
        gridP[Index(N, ibinm, jbin, kbinm)] += hxm * hy0 * hzm * value;
        gridP[Index(N, ibinm, jbin, kbin)] += hxm * hy0 * hz0 * value;
        gridP[Index(N, ibinm, jbin, kbinp)] += hxm * hy0 * hzp * value;
        gridP[Index(N, ibinm, jbinp, kbinm)] += hxm * hyp * hzm * value;
        gridP[Index(N, ibinm, jbinp, kbin)] += hxm * hyp * hz0 * value;
        gridP[Index(N, ibinm, jbinp, kbinp)] += hxm * hyp * hzp * value;

        gridP[Index(N, ibin, jbinm, kbinm)] += hx0 * hym * hzm * value;
        gridP[Index(N, ibin, jbinm, kbin)] += hx0 * hym * hz0 * value;
        gridP[Index(N, ibin, jbinm, kbinp)] += hx0 * hym * hzp * value;
        gridP[Index(N, ibin, jbin, kbinm)] += hx0 * hy0 * hzm * value;
        gridP[Index(N, ibin, jbin, kbin)] += hx0 * hy0 * hz0 * value;
        gridP[Index(N, ibin, jbin, kbinp)] += hx0 * hy0 * hzp * value;
        gridP[Index(N, ibin, jbinp, kbinm)] += hx0 * hyp * hzm * value;
        gridP[Index(N, ibin, jbinp, kbin)] += hx0 * hyp * hz0 * value;
        gridP[Index(N, ibin, jbinp, kbinp)] += hx0 * hyp * hzp * value;

        gridP[Index(N, ibinp, jbinm, kbinm)] += hxp * hym * hzm * value;
        gridP[Index(N, ibinp, jbinm, kbin)] += hxp * hym * hz0 * value;
        gridP[Index(N, ibinp, jbinm, kbinp)] += hxp * hym * hzp * value;
        gridP[Index(N, ibinp, jbin, kbinm)] += hxp * hy0 * hzm * value;
        gridP[Index(N, ibinp, jbin, kbin)] += hxp * hy0 * hz0 * value;
        gridP[Index(N, ibinp, jbin, kbinp)] += hxp * hy0 * hzp * value;
        gridP[Index(N, ibinp, jbinp, kbinm)] += hxp * hyp * hzm * value;
        gridP[Index(N, ibinp, jbinp, kbin)] += hxp * hyp * hz0 * value;
        gridP[Index(N, ibinp, jbinp, kbinp)] += hxp * hyp * hzp * value;
    }

    for (n = 0; n < (N * N * N); n++)
    {
        pout[n] = gridP[n];
    }
    free(gridP);
    return 0;
}


int Index(int N, int x, int y, int z)
{
    return x * N * N + y * N + z;
}
```

cic.c

```python
#!/usr/bin/env python
# coding=utf-8
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef(
    "int CIC(int N, int L, int SIZE, float* posx, float* posy, float* posz, float* pout);")

ffibuilder.set_source(
    "_cic",  # name of the output C extension
    """
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>
    #include <memory.h>
    """,
    sources=['cic.c'],   # includes cic.c as additional sources
    include_dirs=['./'],
    #   library_dirs=[],
    libraries=['m'])    # on Unix, link with the math library

if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

libgrid_build.py