kdeLF

Release 1.3.0

unknown

Jul 06, 2022

USER GUIDE

kdeLF is an MIT licensed Python implementation of Yuan et al.'s Kernel Density Estimation (KDE) method for estimating luminosity functions and these pages will show you how to use it.

This documentation won't teach you too much about KDE but there are a lot of resources available for that (try this one). Our paper explaining the kdeLF algorithm and implementation in detail. kdeLF is being actively developed on GitHub.

CHAPTER

ONE

BASIC USAGE

If you want to calculate the luminosity function based on a survey data, you would do something like:

```
import numpy as np
from kdeLF import kdeLF
from scipy import interpolate

with open('flim.dat', 'r') as f:
    x0, y0= np.loadtxt(f, usecols=(0,1), unpack=True)
f_lim = interpolate.interp1d(x0, y0,fill_value="extrapolate")
sr=6248/((180/np.pi)**2)

lf=kdeLF.KdeLF(sample_name='data.txt',solid_angle=sr,zbin=[0.6,0.8],f_lim=f_lim,
↪adaptive=True)
lf.get_optimal_h()
lf.run_mcmc()
lf.plot_posterior_LF(z=0.718,sigma=3)
```

A more complete example is available in the *Quickstart* tutorial.

CHAPTER

TWO

HOW TO USE THIS GUIDE

To start, you're probably going to need to follow the *Installation* guide to get kdeLF installed on your computer. After you finish that, you can probably learn most of what you need from the tutorials listed below (you might want to start with *Quickstart* and go from there). If you need more details about specific functionality, the User Guide below should have what you need.

We welcome bug reports, patches, feature requests, and other comments via the GitHub issue tracker, but you should check out the contribution guidelines first.

2.1 Installation

2.1.1 Using pip

The recommended way to install the stable version of kdeLF is using pip

```
pip install -U kdeLF
```

We have uploaded several *.whl* files to the PyPI web to support as many platforms as possible. If your platforms happens to be an exception, then the pip installation may fail. In this situation, you need to install the Intel fortran Compiler first, and then try the pip installation again. Note that the version of NumPy library should 1.21.0. If you use the latest Intel fortran Compiler, then a higher version (1.22) of NumPy is required. If you have problems installing, please open an issue at GitHub.

2.1.2 From source

You can also install *kdeLF* after a download from GitHub. Note that this requires a Intel fortran Compiler to be installed in advance.

```
git clone https://github.com/yuanzunli/kdeLF.git
cd kdeLF
pip install .
```

2.1.3 Test the installation

To make sure that the installation went alright, you can execute the built-in test program in kdeLF by running the following command:

```
python3 -m kdeLF.test_kdeLF
```

2.2 The class KdeLF

Standard usage of `kdeLF` involves instantiating an `KdeLF`.

coming soon…

2.3 FAQ

The not-so-frequently asked questions that still have useful answers

coming soon …

2.4 Quickstart

To get you started, here's an annotated, fully-functional example that demonstrates a standard usage pattern for kdeLF. I will generate a synthetic dataset (or called simulated sample) from a known luminosity function (LF), and then apply kdeLF to it.

2.4.1 Monte Carlo sampling from a known LF

The differential LF of a sample of objects is defined as the number of objects per unit comoving volume per unit luminosity interval,

$$\phi(\mathcal{L}, z) = \frac{d^2N}{dV d\mathcal{L}},$$

where z denotes redshift and $\mathcal{L}$ denotes the luminosity. Due to the typically large span of the luminosities, it is often defined in terms of $\log \mathcal{L}$,

$$\phi(L, z) = \frac{d^2N}{dV dL},$$

where $L \equiv \log \mathcal{L}$ denotes the logarithm of luminosity. Our input LF has a pure density evolution form, with the density evolution function of

$$e(z) = p_0 \left[\left(\frac{1+z_c}{1+z} \right)^{p_1} + \left(\frac{1+z_c}{1+z} \right)^{p_2} \right]^{-1},$$

where p_0 is the normalized parameter of $e(z)$ given by

$$p_0 = \left[(1+z_c)^{p_1} + (1+z_c)^{p_2} \right],$$

and

$$\phi(L,z) = n_0 e(z) \left(\frac{10^L}{L_*}\right)^{-\alpha} \exp\left[-\left(\frac{10^L}{L_*}\right)^{\beta}\right].$$

In the above equations, z_c=2.0, p_1=1.5, p_2=−5, L_*=10^{25}, n_0=10^{-5}, α=0.75, β=0.3, are free parameters. The users may change the parameter values, or even provide a quite different form of input LF for the Monte Carlo sampling.

The first thing that we need to do is import the necessary modules:

```
import matplotlib.pyplot as plt
import numpy as np
import math
from scipy.integrate import dblquad
from scipy.integrate import quad
from scipy.optimize import minimize_scalar
from tqdm import tqdm
```

Then, we'll code up a Python function that returns the density evolution function $e(z)$ and the input (true) LF $\phi(L,z)$, called `phi_true`:

```
zc,p1,p2,L0,n0,alpha,beta=2.0,2.0,-4.5,25.0,-5.0,0.75,0.30

def e(z):
    rho=1.0/( ((1+zc)/(1+z))**p1 + ((1+zc)/(1+z))**p2 ) * ( ((1+zc)/(1+0.0))**p1 +␣
↪((1+zc)/(1+0.0))**p2)
    return rho

def phi_true(z,L):
    L_z=10**L0
    rho=10**n0 * (10**L/L_z)**(-alpha)*np.exp(-(10**L/L_z)**beta) * e(z)
    return rho
```

Then, we'll code up a Python function that returns the truncation boundary $f_{\rm lim}(z)$ of the sample, called `f_lim`. Here we consider the case for the common flux-limited samples, $f_{\rm lim}(z)$ is given by

$$f_{\rm lim}(z) = 4\pi d_L^2(z)(1/K(z))F_{\rm lim},$$

where $d_L(z)$ is the luminosity distance, $F_{\rm lim}$ is the survey flux limit, and $K(z)$ represents the K-correction.

```
flux,spectral_index= 0.001,0.75
H0,Om0 = 70, 0.3
Mpc=3.08567758E+022
c=299792.458
from astropy.cosmology import FlatLambdaCDM
cosmo = FlatLambdaCDM(H0=H0, Om0=Om0)
## your own f_lim function :
def f_lim(z):
    St=flux
    dist=cosmo.luminosity_distance(z)
    ans=dist.value
    return np.log10(4*np.pi*(ans*Mpc)**2*St*1E-26/(1.0+z)**(1.0-spectral_index))
```

Set the redshift and luminosity ranges, as well as the desired sample size for the simulated sample:

```
z1,z2,L1,L2=0.0,6.0,22.0,30.0
ndata=8000
#The actual sample size of the simulated sample will not exactly equals to ndata, but␣
↪should be very close.
```

The following code will estimate the solid angle omega (unit of sr) of the simulated sample:

```
def rho(L,z):
    dvdz=cosmo.differential_comoving_volume(z).value
    result=phi_true(z,L)*dvdz
    return result

ans=dblquad(rho, z1, z2, lambda z: L1, lambda z: L2)
ans_lim=dblquad(rho, z1, z2, lambda z: f_lim(z), lambda z: L2)
omega=ndata/ans_lim[0]
N_tot=math.ceil(ans[0]*omega)
print("The solid angle 'omega' of the simulated sample:",'%.4f' % omega)
```

```
The solid angle 'omega' of the simulated sample: 0.0120
```

We then code up the marginal probability density distributions (PDFs) for z and L, and prepare for a Monte Carlo sampling:

```
def pz(z):              # the PDF of z, not normalized
    dvdz=cosmo.differential_comoving_volume(z).value
    return e(z)*dvdz

def pl(L):          # the PDF of L, not normalized
    L_z=10**L0
    result= (10**L/L_z)**(-alpha)*np.exp(-(10**L/L_z)**beta)
    return result

ans=quad(pz,z1,z2)
nomz=ans[0]
ans=quad(pl,L1,L2)
noml=ans[0]
#print('nomz,noml',nomz,noml)

res = minimize_scalar(lambda z: -pz(z)/nomz, bounds=(z1, z2), method='bounded')
zmp=res.x
max_pz=-res.fun
#print('zmax',zmp,max_pz)
res = minimize_scalar(lambda L: -pl(L)/noml, bounds=(L1, L2), method='bounded')
lmp=res.x
max_pl=max(-res.fun,pl(L1)/noml)
#print('Lmax',lmp,max_pl)
Lambta_z=1.0/max_pz
Lambta_L=1.0/max_pl
```

Start sampling with the rejection method

```
from random import seed, random
from multiprocessing import Pool, cpu_count
```

(continues on next page)

(continued from previous page)

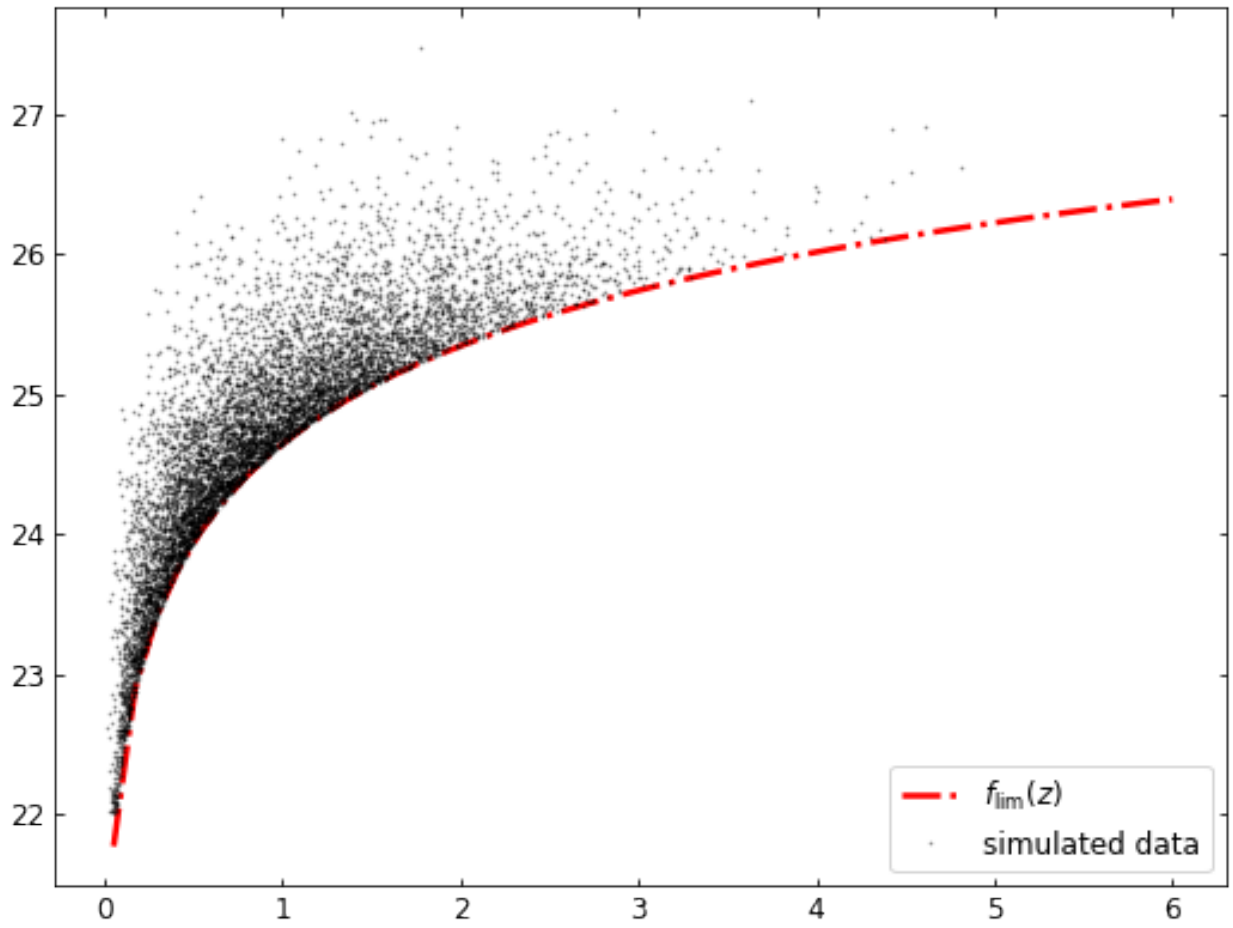
```
cores = cpu_count()-1
N_simu=math.ceil(N_tot/cores)

def sampler(ir):
    rst=[]
    if ir==0:
        irs=tqdm(range(N_simu))
    else:
        irs=range(N_simu)
    for i in irs:
        seed()
        while True:
            si1=random()
            Delta_z=z1+(z2-z1)*si1
            si2=random()
            Judge_z=Lambta_z*pz(Delta_z)/nomz
            if si2<=Judge_z:
                z_simu=Delta_z
                break

        while True:
            Xi_L1=random()
            Delta_L=L1+(L2-L1)*Xi_L1
            Xi_L2=random()
            Judge_L=Lambta_L*pl(Delta_L)/noml
            if Xi_L2<=Judge_L:
                L_simu=Delta_L
                break

        Llim=f_lim(z_simu)
        if L_simu>Llim:
            rst.append((z_simu,L_simu))
    return rst

ps=np.arange(cores)

if __name__ == '__main__':
    pool = Pool(cores) # Create a multiprocessing Pool
    rslt = pool.map(sampler, ps)  # process data_inputs iterable with pool
    pool.close()
    pool.join()

f = open('data.txt', "w")
for i in range(len(rslt)):
    for j in range(len(rslt[i])):
        print('%.6f' %rslt[i][j][0],'%.6f' %rslt[i][j][1],file=f)
f.close()
```

```
100%|| 272350/272350 [03:54<00:00, 1161.95it/s]
```

The synthetic data has been saved to to `data.txt`. The following code will plot the synthetic data:

```
with open('data.txt', 'r') as f:
    red,lum= np.loadtxt(f, usecols=(0,1), unpack=True)
print('total number:',red.size)
zs=np.linspace(0.05,6)
ls=f_lim(zs)
plt.figure(figsize=(8,6))
ax=plt.axes([0.1,0.1, 0.85, 0.85])
ax.tick_params(direction='in', top=True, right=True, labelsize=12)
plt.plot(zs,ls,'-.',color=(1.0,0.0,0.0),linewidth=2.5, label=r'$f_{\mathrm{lim}}(z)$')
plt.plot(red,lum,'.',ms=1.2,color=(0.0,0.0,0.0),label='simulated data',alpha=0.4)
ax.legend(fontsize=12,loc='lower right')
plt.show()
```

```
total number: 7873
```



2.4.2 Apply kdeLF to the simulated sample

Set the range of redshift. Generally we let Z1 be slightly less than the minimum redshift of sample, and Z2 be slightly greater than the maximum redshift:

```
print(min(red),max(red))
```

```
0.014599 4.810231
```

```
Z1, Z2 =0.01,4.90
```

The main interface provided by kdeLF is the `KdeLF` object. It acts as the interface that receives the input of required and optional arguments to initialize a calculation. The following code will initialize a KdeLF instance, named "lf1":

```
from kdeLF import kdeLF

lf1 = kdeLF.KdeLF(sample_file='data.txt', solid_angle=omega, zbin=[Z1,Z2], f_lim=f_lim,
                  H0=H0, Om0=Om0, small_sample=False, adaptive=False)
```

The first four arguments are required and the others are optional. `sample_file` is the name of sample file that contains at least two columns of data for z and L (or absolute magnitude M). `zbin` is the redshift range $[Z_1, Z_2]$. `f_lim` is the user defined Python function calculating the truncation boundary $f_{\rm lim}(z)$ of sample, and `solid_angle` is the solid angle (unit of sr) subtended by the sample. kdeLF adopts a Lambda Cold Dark Matter (LCDM) cosmology, but it is not limited to this specific cosmological model. The optional arguments `Om0` and `H0`, defaulting as 0.30 and 70 (km $\rm s^{-1}$ $\rm Mpc^{-1}$), represent the Ω_m parameter and Hubble constant for the LCDM cosmology, respectively.

In our paper (Yuan et al. 2022), our KDE method provided **four LF estimators. We denote the LF estimated by Equation (15) in our paper as $\hat{\phi}$, and the small sample approximation by Equation (25) as $\hat{\phi}_1$. Their adaptive versions are denoted as $\hat{\phi}_{\rm a}$ and $\hat{\phi}_{1\rm a}$, respectively**. The other two optional arguments of `KdeLF`, `small_sample` and `adaptive`, have four different combinations, (False, False) , (False, True) , (True, False) and (True, True) , corresponding to the usages of $\hat{\phi}$, $\hat{\phi}_{\rm a}$, $\hat{\phi}_1$ and $\hat{\phi}_{1\rm a}$, respectively.

After the above initialization, kdeLF is ready for calculating the LF using the $\hat{\phi}$ estimator. We can obtain the optimal bandwidths of KDE by the `KdeLF.get_optimal_h ()` function:

```
lf1.get_optimal_h()
```

```
z & L data loaded
Maximum likelihood estimation by scipy.optimize 'Powell' method,
redshift bin: ( 0.01 , 4.9 )
sample size of this bin: 7872
bandwidths for 2d estimator,
    Initial h1 & h2:      [0.15, 0.15]
    bounds for h1 & h2:   [(0.001, 1.0), (0.001, 1.0)]
    Optimal h1 & h2:
         0.3019 0.1059
Cost total time: 7.72  second
```

```
array([0.3019075 , 0.10594656])
```

Once having the optimal bandwidths, we can get the LF estimates at any point (z, L) in the in the domain of $\{Z_1 < z < Z_2,\ L > f_{\rm lim}(z)\}$, e.g.,

```
lf1.phi_kde(0.5,25.0,[0.32395053, 0.10842562])
```

```
7.809333168741918e-06
```

More often, we are interested in plotting the LF at some given redshift, which can be achieved by the `KdeLF.get_lgLF()` function. This will return a two-dimensional array giving the mesh points for L and the LF estimates.

```
lfk=lf1.get_lgLF(z=0.5,plot_fig=False)
```

For comparison, we also try the adaptive KDE method: $\hat{\phi}_{\rm a}$:

```
lf2 = kdeLF.KdeLF(sample_file='data.txt', solid_angle=omega, zbin=[Z1,Z2], f_lim=f_lim,
                  H0=H0, Om0=Om0, small_sample=False, adaptive=True)

lf2.get_optimal_h()
```

```
z & L data loaded
Maximum likelihood estimation by scipy.optimize 'Powell' method,
redshift bin: ( 0.01 , 4.9 )
sample size of this bin: 7872
bandwidths for 2d estimator,
    Initial h1 & h2:      [0.15, 0.15]
    bounds for h1 & h2:   [(0.001, 1.0), (0.001, 1.0)]
    Optimal h1 & h2:
         0.3019 0.1059
pilot bandwidths:
    h1p & h2p: 0.3019 0.1059

global bandwidths and beta for adaptive 2d estimator,
    Initial h10, h20 & beta:      [0.15 0.15 0.3 ]
    bounds for h10, h20 & beta:   [(0.001, 1.0), (0.001, 1.0), (0.01, 1.0)]
    Optimal h10, h20 & beta:
         0.1151 0.0606 0.2505

Cost total time: 21.92  second
```

```
array([0.1151027 , 0.06057506, 0.25046887])
```

```
lfka=lf2.get_lgLF(z=0.5,plot_fig=False)
```

It would be valuable to compare our KDE method with the classical binning estimator. kdeLF provides the `KdeLF.get_binLF ()` function to get the LF estimates using Page & Carrera (2000) 's binning estimator, denoted as $\hat{\phi}_{\rm bin}$.

```
blf=lf1.get_binLF(Lbin=0.3,zbin=[0.3,0.7],plot_fig=False)
```

This will return a tuple containing the arrays of L and the LF estimates, as well as their errors. Then we plot the LFs obtained by different estimators:

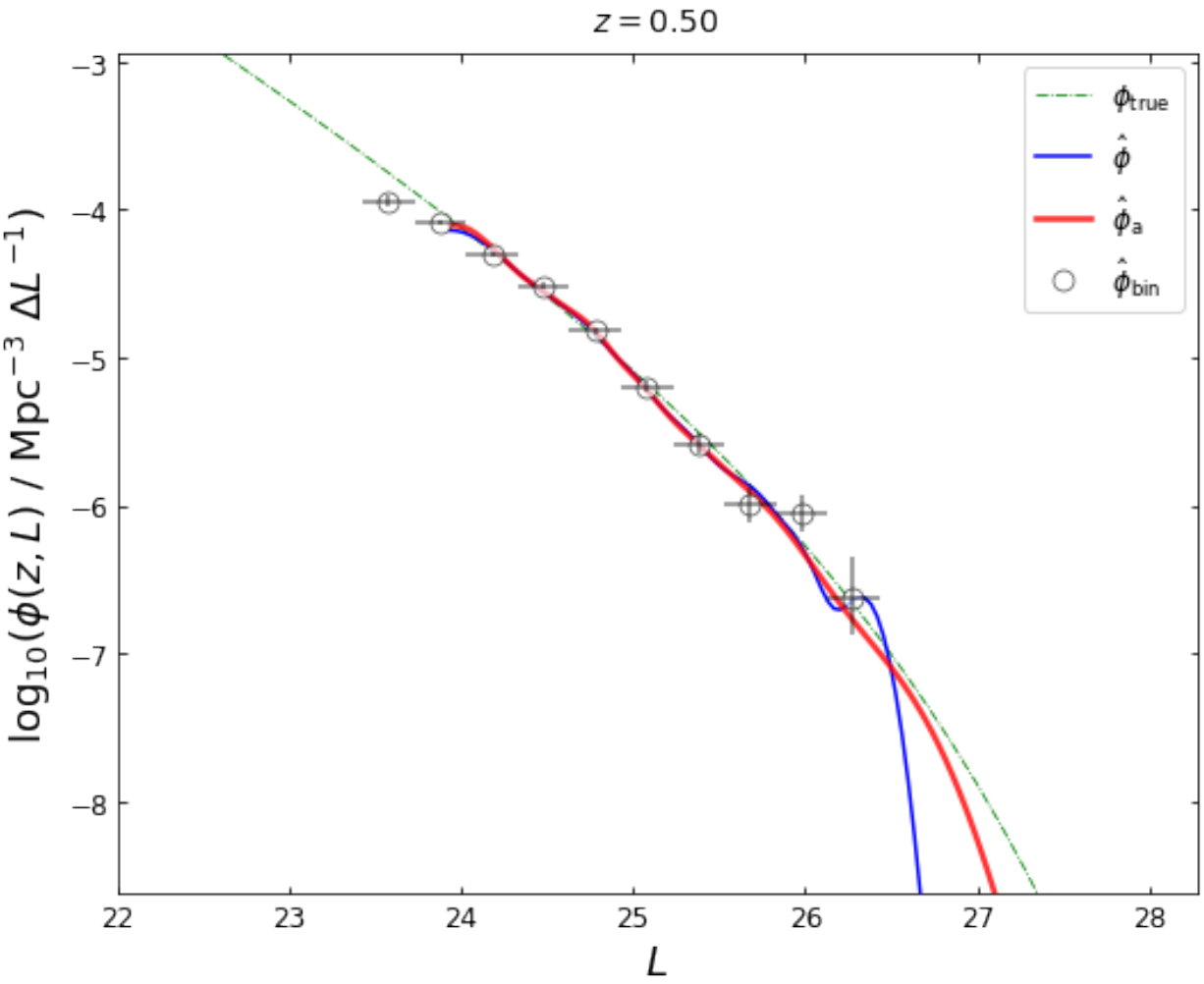
```
def plot_lfs(z_plot,lfk=None,lfka=None,blf=None,lfk1a=None):
    lum=np.linspace(22.0,29.0)
    plt.figure(figsize=(8,6))
```

(continues on next page)

(continued from previous page)

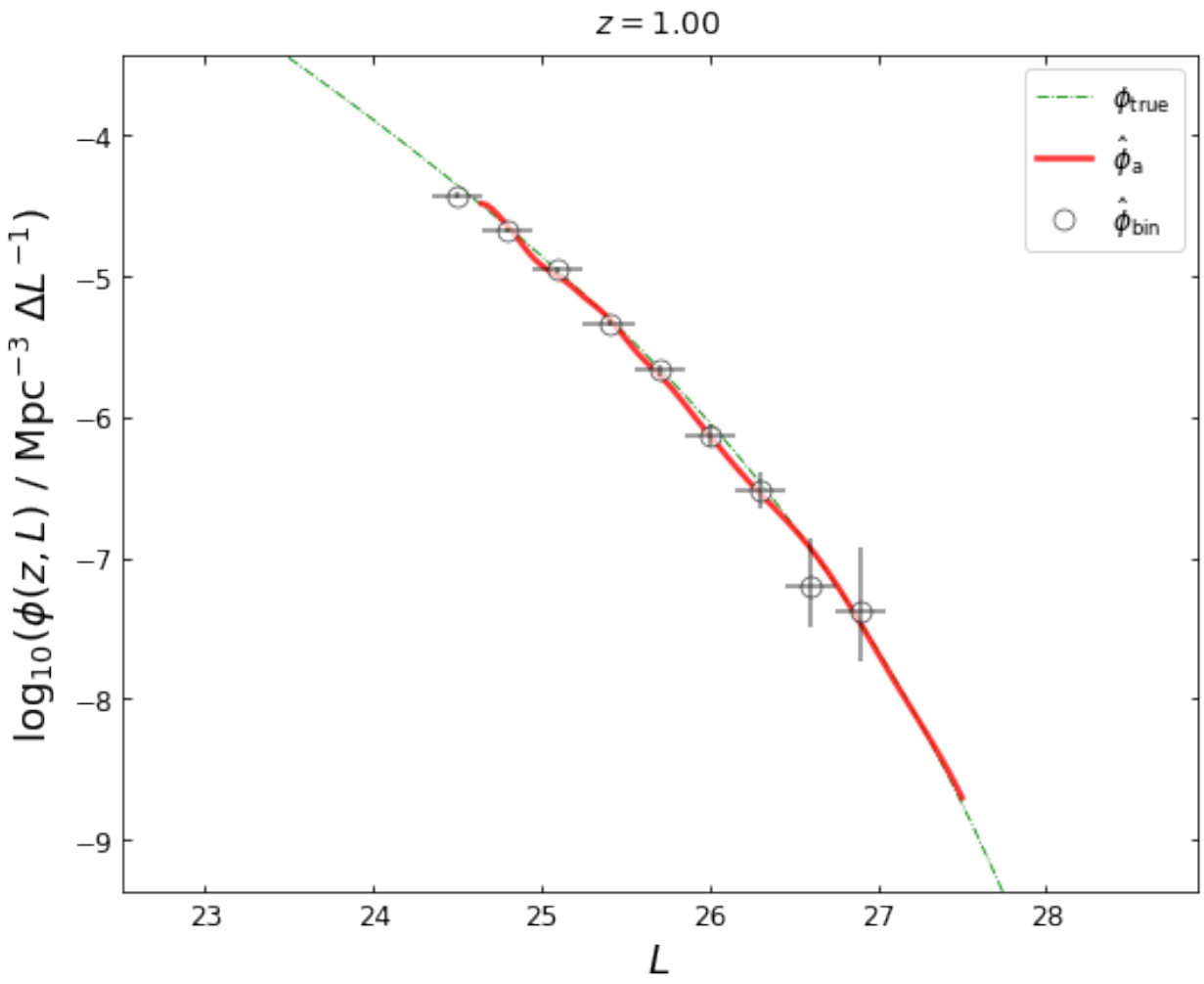
```
    ax=plt.axes([0.13,0.1, 0.82, 0.85])
    ax.tick_params(direction='in', top=True, right=True, labelsize=12)
    phi=np.log10(phi_true(z_plot,lum))
    plt.plot(lum,phi,linestyle='-.',color='green',linewidth=0.8, label=r'$\phi_{\mathrm
↪{true}}$')
    if lfk is not None:
        plt.plot(lfk[0],lfk[1],color=(0.0,0.0,1.0),linewidth=1.5, label=r'$\hat{\phi}$')
    if lfka is not None:
        plt.plot(lfka[0],lfka[1],color='red',linewidth=2.5, alpha=0.8, label=r'$\hat{\
↪phi}_{\mathrm{a}}$')
    if lfk1a is not None:
        plt.plot(lfk1a[0],lfk1a[1],color='cyan',linewidth=2.5, alpha=0.8, label=r'$\hat{\
↪phi}_{\mathrm{1a}}$')
    if blf is not None:
        plt.plot(blf[0],blf[3],'o',mfc='white',mec='black',ms=9,mew=0.7,alpha=0.7,label=r
↪'$\hat{\phi}_{\mathrm{bin}}$')
    ax.errorbar(blf[0],blf[3], ecolor='k', capsize=0,
                xerr=np.vstack((blf[1], blf[2])),
                yerr=np.vstack((blf[4], blf[5])),
                fmt='None', zorder=4,alpha=0.5)
    plt.ylabel(r'$\log_{10}( \phi(z,L) ~/~ {\rm Mpc}^{-3} ~ \Delta L^{-1} )$',
↪fontsize=18)
    ax.legend(fontsize=12,loc='upper right')
    plt.xlabel(r'$L$',fontsize=18)
    plottitle = r'$z={0:.2f}$'.format(z_plot)
    plt.title(plottitle, size=14, y=1.01)
    if blf is not None:
        x1,x2=max((min(blf[0])-2), min(lum)), min((max(blf[0])+2), max(lum))
        y1,y2=min(blf[3])-2,max(blf[3])+1
        plt.xlim(x1,x2)
        plt.ylim(y1,y2)
    plt.show()
```

```
plot_lfs(z_plot=0.5,lfk=lfk,lfka=lfka,blf=blf)
```

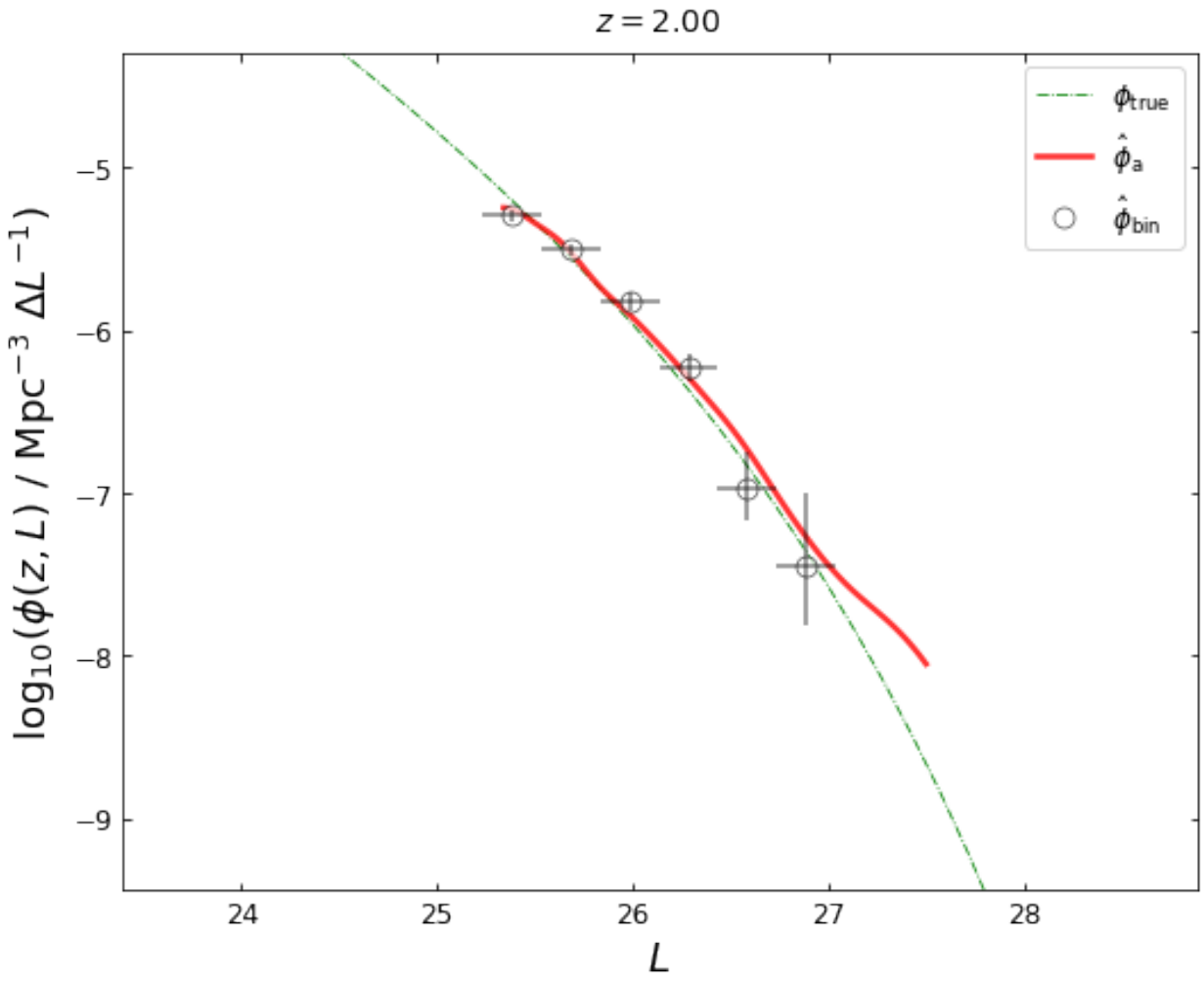


We can plot the LFs at any redshift in the interval of (Z_1, Z_2)

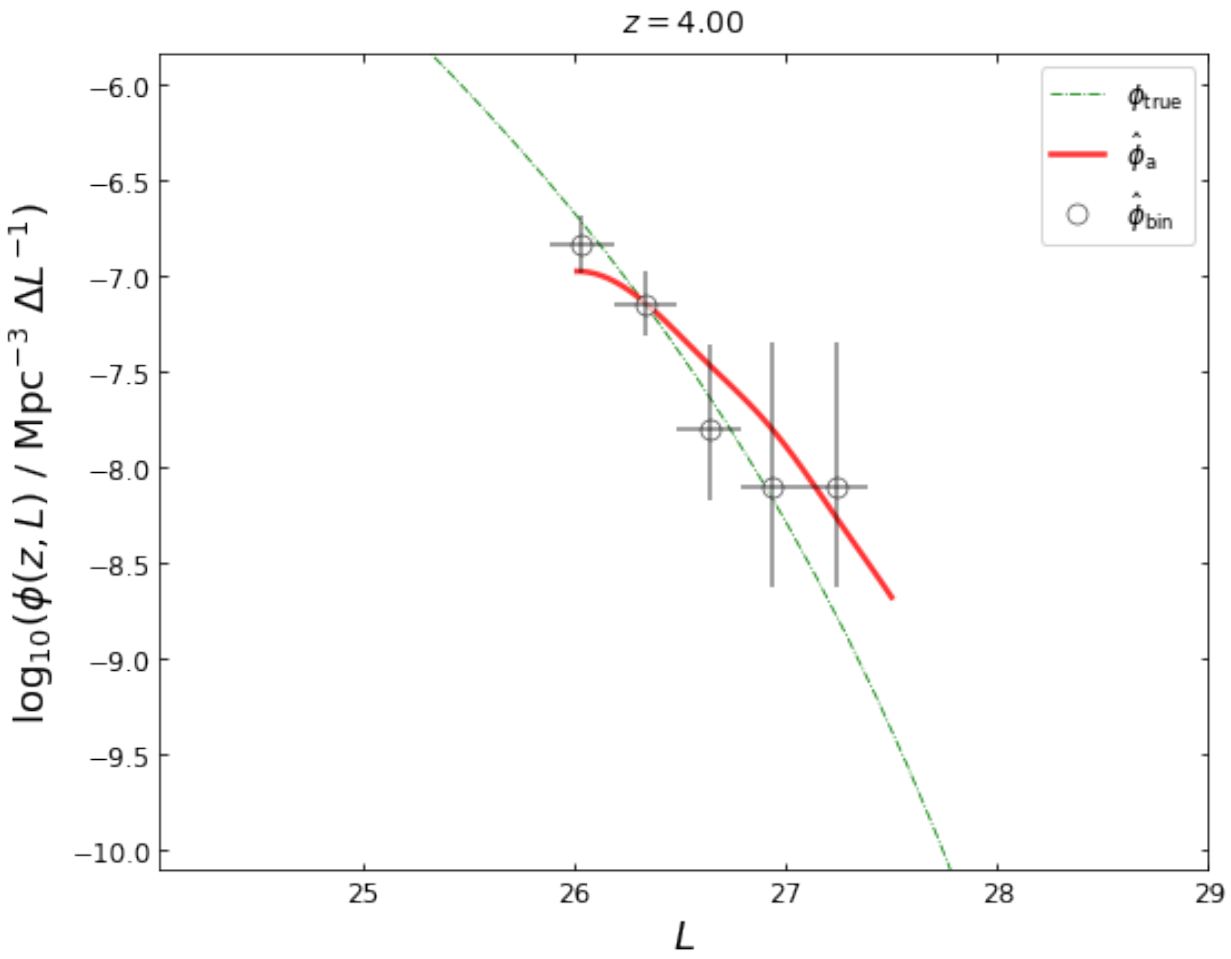
```
z_plot=1.0
blf=lf1.get_binLF(Lbin=0.3,zbin=[0.75,1.25],plot_fig=False)
lfka=lf2.get_lgLF(z=z_plot,plot_fig=False)
plot_lfs(z_plot=z_plot,lfka=lfka,blf=blf)
```




```
z_plot=2.0
blf=lf1.get_binLF(Lbin=0.3,zbin=[1.8,2.2],plot_fig=False)
lfka=lf2.get_lgLF(z=z_plot,plot_fig=False)
plot_lfs(z_plot=z_plot,lfka=lfka,blf=blf)
```




```
z_plot=4.0
blf=lf2.get_binLF(Lbin=0.3,zbin=[3.5,4.5],plot_fig=False)
lfka=lf2.get_lgLF(z=z_plot,plot_fig=False)
plot_lfs(z_plot=z_plot,lfk=None,lfka=lfka,blf=blf)
```



We note that the LF at z=4 given by $\hat{\phi}_a$ is not good, mainly because the sources near z~4 is very sparse.

```
select=((red>3.5) & (red<4.5))
print('number:',len(red[select]))
```

```
number: 24
```

There are only 24 sources in the (3.5,4.5) redshift bin. For this small sample case, a better solution is using the small sample approximation KDE method:

```
lf3=kdeLF.KdeLF(sample_file='data.txt', solid_angle=omega, zbin=[3.5,4.5], f_lim=f_lim,
                H0=H0, Om0=Om0, adaptive=True, small_sample=True)
lf3.get_optimal_h()
```

```
z & L data loaded
Maximum likelihood estimation by scipy.optimize 'Powell' method,
redshift bin: ( 3.5 , 4.5 )
sample size of this bin: 24
bandwidth for 1d estimator,
    bound for h:   (0.001, 1.0)
    Optimal h:     0.2928
pilot bandwidth:
```

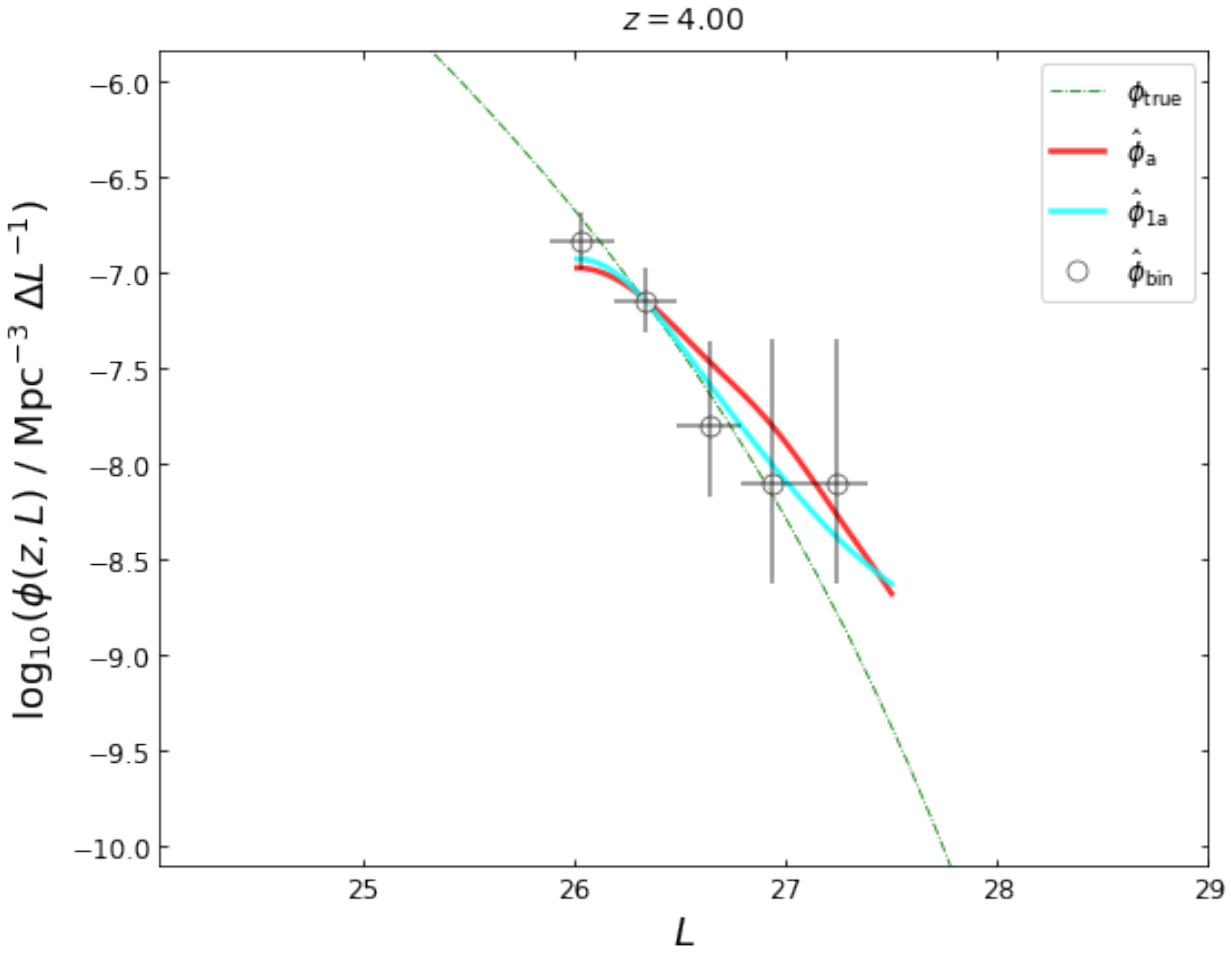
(continues on next page)

(continued from previous page)

```
    hp: 0.2928

global bandwidth and beta for adaptive 1d estimator,
    Initial h0 & beta:      [0.29283326 0.3       ]
    bounds for h0 & beta:   [(0.001, 1.0), (0.01, 1.0)]
    Optimal h0 & beta:
         0.3916 1.0000
```

```
array([0.3916132, 1.       ])
```

```
z_plot=4.0
lfk1a=lf3.get_lgLF(z=z_plot,plot_fig=False)
blf=lf3.get_binLF(Lbin=0.3,zbin=[3.5,4.5],plot_fig=False)
plot_lfs(z_plot=z_plot,lfk1a=lfk1a,lfka=lfka,blf=blf)
```



2.4.3 MCMC & uncertainty estimation

The `KdeLF.get_optimal_h()` function can only give the "best-fit" values for the bandwidths. If you want an uncertainty estimation to the bandwidths, you can use the `KdeLF.run_mcmc()` function. It implements a fully Bayesian Markov Chain Monte Carlo (MCMC) method to determine the posterior distributions of bandwidths and other parameters (e.g., β for adaptive KDE). The MCMC core embedded in kdeLF is the Python package emcee. `KdeLF.run_mcmc()` uses the ensemble MCMC sampler provided by emcee to draw parameter samples from the posterior probability distribution (the adaptive KDE $\hat{\phi}_{\rm a}$ for example):

$$p(h_{10}, h_{20}, \beta \,|\, z_{\rm i}, L_{\rm i}) \propto p(h_{10}, h_{20}, \beta)\, p(z_{\rm i}, L_{\rm i} \,|\, h_{10}, h_{20}, \beta) \quad ,$$

where $p(z_{\rm i}, L_{\rm i} \,|\, h_{10}, h_{20}, \beta)$ is the likelihood function (see Section 2.3 of (Yuan et al. 2022) for details). Its calculation is completed automatically by the program. $p(h_{10}, h_{20}, \beta)$ is the prior distribution, and by default, we use uniform (so-called "uninformative") priors on h_{10}, h_{20}, and β.

```
lfmc = kdeLF.KdeLF(sample_file='data.txt', solid_angle=omega, zbin=[Z1,Z2], f_lim=f_lim,
                   H0=H0, Om0=Om0, small_sample=False, adaptive=True)
lfmc.get_optimal_h(quiet=True)
```

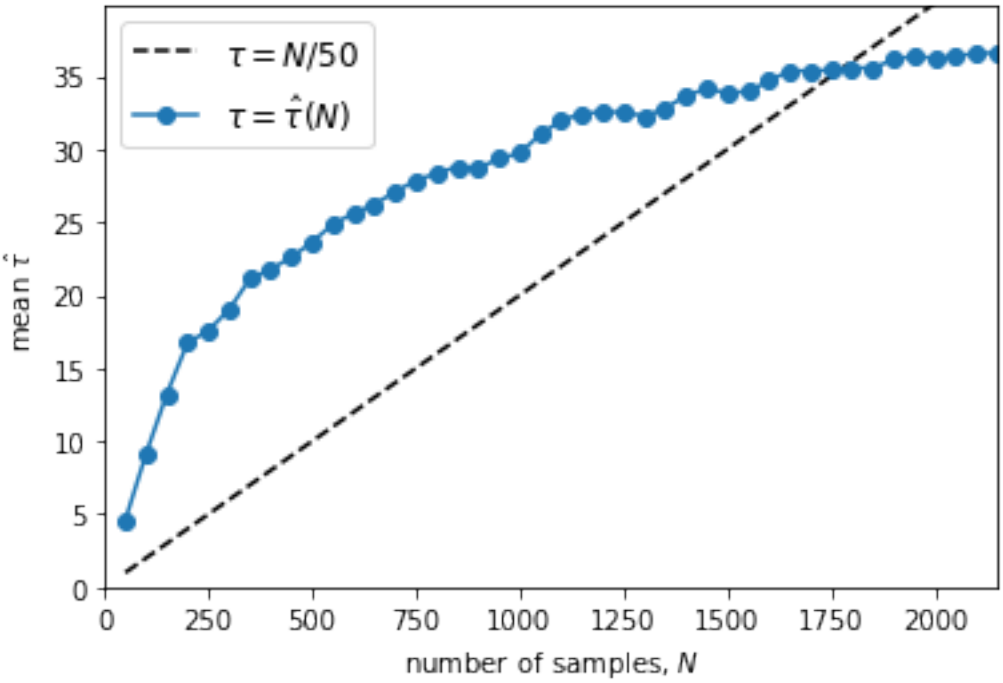
```
z & L data loaded
```

```
array([0.1151027 , 0.06057506, 0.25046887])
```

```
lfmc.run_mcmc()
# full parameter usage: lfmc.run_mcmc(max_n=3000, Ntau=50, initial_point=None,
#                                     parallel=False, priors=None, chain_analysis=False)
```

```
 72%|          | 2150/3000 [6:02:31<2:23:19, 10.12s/it]
```

```
burn-in: 84
thin: 16
flat chain shape: (4128, 3)
flat log prob shape: (4128,)
```

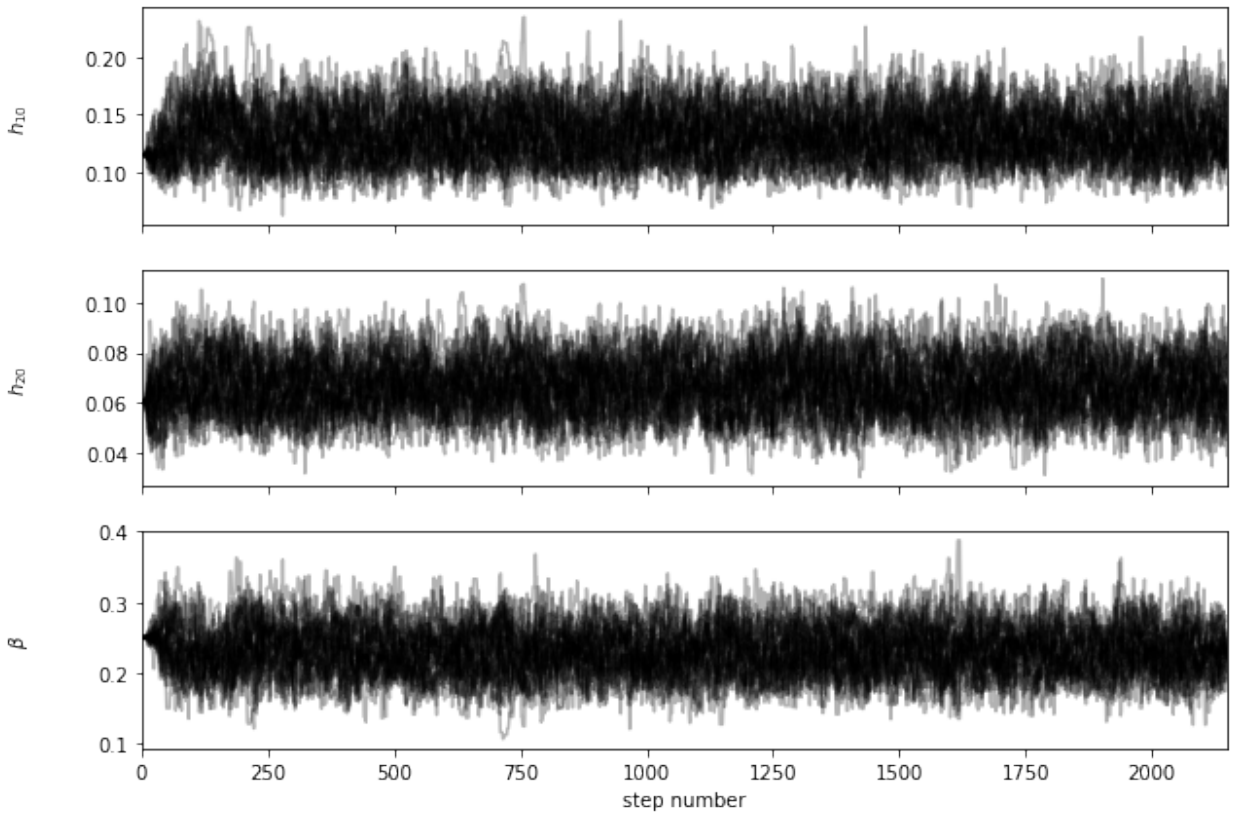


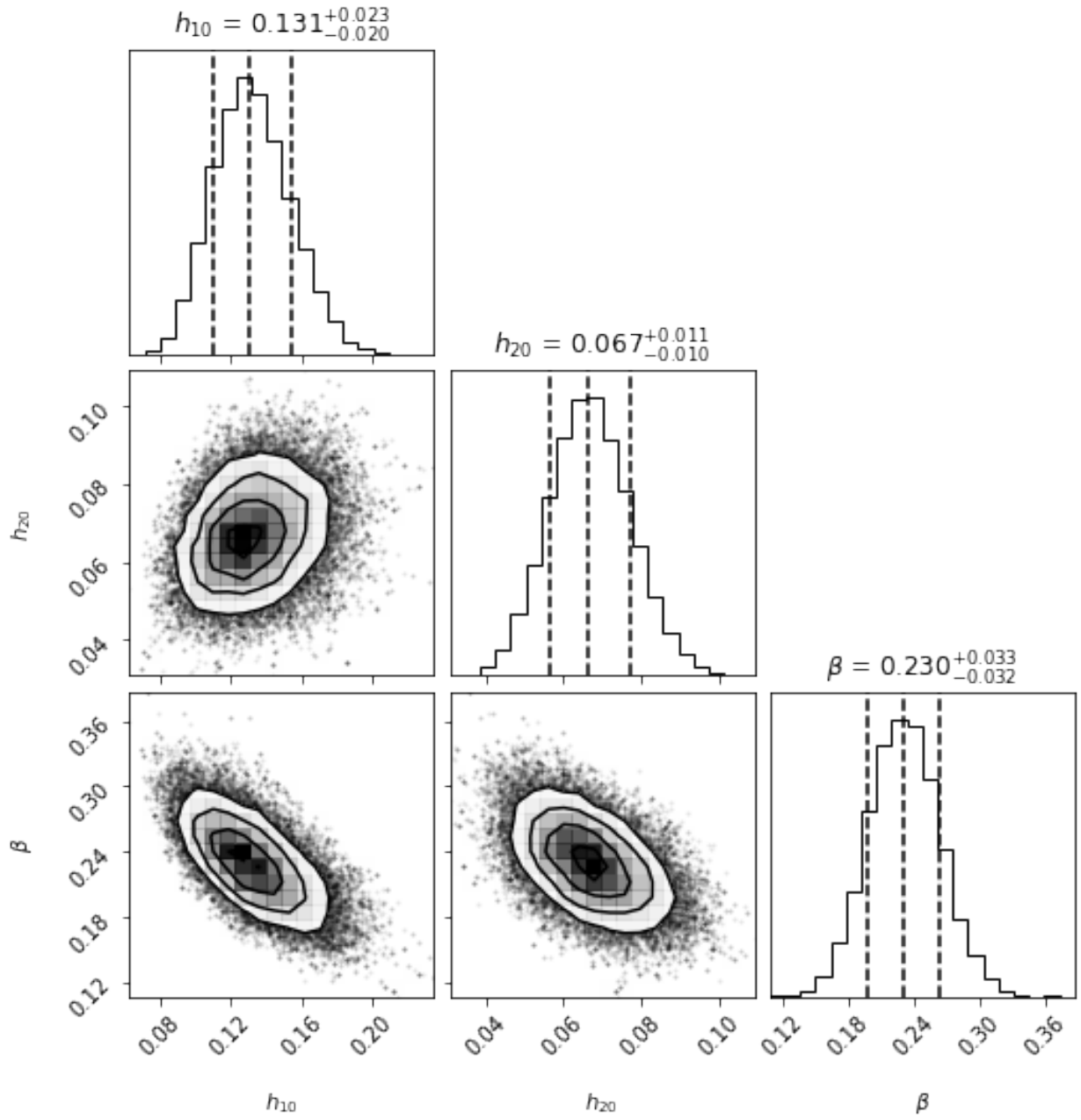
Running the above code is generally time-consuming. It may take several minutes to tens hours, depending on the sample size and your CPU performance. The program incrementally save the state of the chain to a HDF5 file (suppose you have the h5py library installed). The code will run the chain for up to `max_n` (default=3000) steps and check the autocorrelation time every 50 steps. If the chain is longer than `Ntau` (default=50) times the estimated autocorrelation time and if this estimate changed by less than 1%, we'll consider it converged. Larger values of `Ntau` will be more conservative, but they will also require longer chains (see the emcee document for details).

A successful running will produce a figure showing the autocorrelation time estimate as a function of chain length. If the chain converges, the blue curve should cross the black dashed line. You will also find a '.h5' file like 'chain_a0.01_4.9.h5' (automatically named by the program) in the current folder. Then we use the `KdeLF.chain_analysis()` function to visualize the chain:

```
lfmc.chain_analysis('chain_a0.01_4.9.h5')
```

```
burn-in: 84
thin: 16
flat chain shape: (4128, 3)
flat log prob shape: (4128,)

********** MCMC bestfit and 1 sigma errors **********
 h10 = 0.1307 - 0.0205 + 0.0233
 h20 = 0.0666 - 0.0102 + 0.0109
beta = 0.2302 - 0.0323 + 0.0331
```


h_{10}
0.20
0.15
0.10
h_{20}
0.10
0.08
0.06
0.04
β
0.4
0.3
0.2
0.1
0
250
500
750
1000
1250
1500
1750
2000
step number



Now we can get the posterior distributions for LFs using the `KdeLF.plot_posterior()` function. This will return a tuple containing the arrays of L, the posterior median LFs, and the errors.

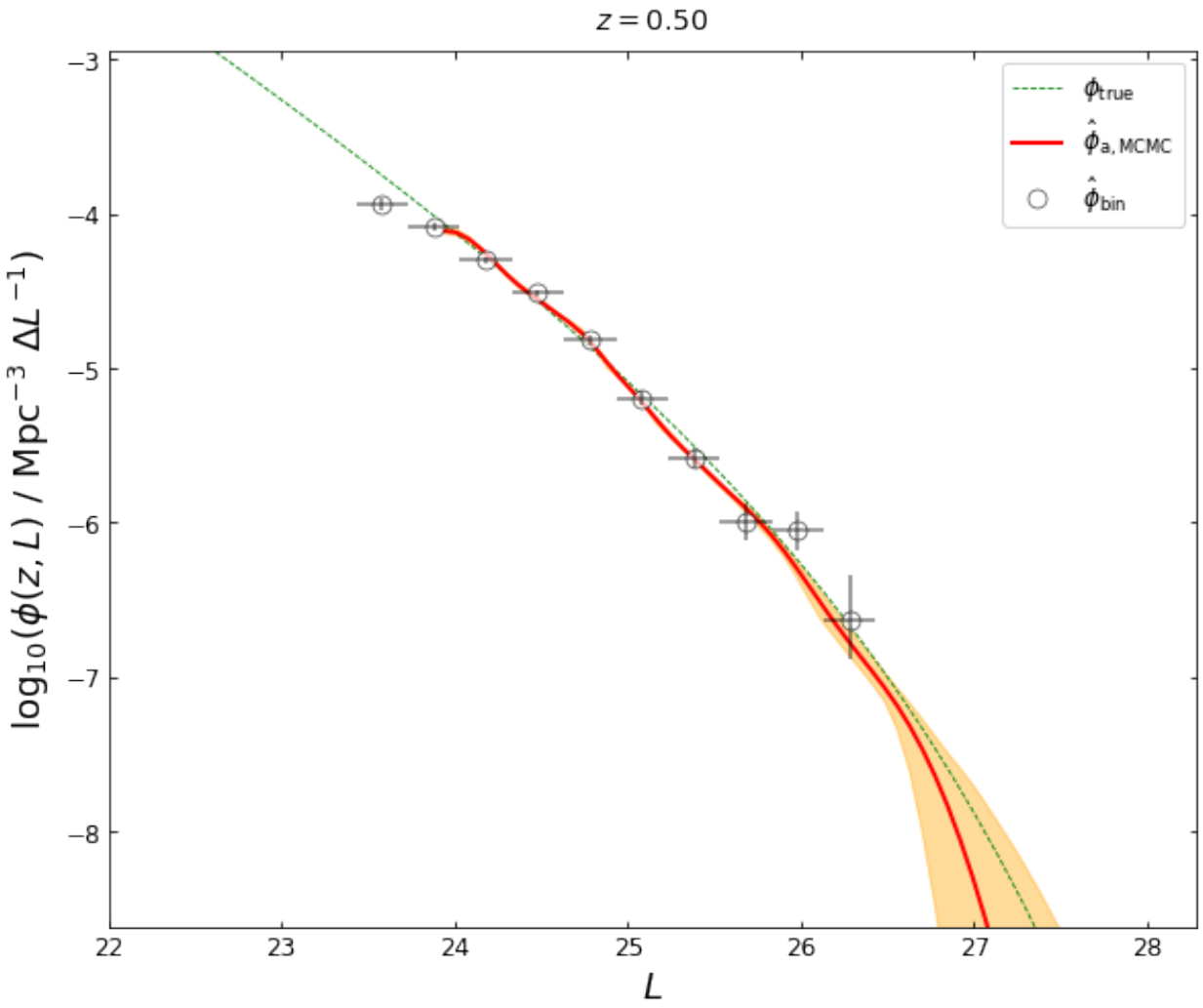
```
lf0_5 = lfmc.plot_posterior_LF(z=0.5,sigma=3,plot_fig=False)
lums, phi_mcmc, lower_error, upper_error = lf0_5
```

```
100%|| 1500/1500 [00:22<00:00, 66.90it/s]
```

Now let's plot the result. The red curve shows our MCMC best-fit LF, which is represented by the median of the posterior probability distribution function. The orange shaded area shows the 3σ uncertainty (99.93% equal-tailed credible interval).

```
def plot_lfs_mcmc(z_plot, blf, lf_mcmc):
    lums, phi_mcmc, lower_error, upper_error = lf_mcmc
    lum=np.linspace(22.0,29.0)
    phi=np.log10(phi_true(z_plot,lum))
    plt.figure(figsize=(9,7))
    ax=plt.axes([0.13,0.1, 0.82, 0.85])
    ax.tick_params(direction='in', top=True, right=True, labelsize=12)
    plt.plot(lum,phi,linestyle='--',color='green',linewidth=0.8, label=r'$\phi_{\mathrm
↪{true}}$')

    ###############  plot the posterior distributions for LFs ##############
    ax.fill_between(lums, lower_error, y2=upper_error, color='orange', alpha=0.4)
    ax.plot(lums, phi_mcmc, lw=2.0, c='red', label=r'$\hat{\phi}_{\mathrm{a,MCMC}}$')

     ###############  plot the binning LFs with errorbars ###################
    plt.plot(blf[0],blf[3],'o',mfc='white',mec='black',ms=9,mew=0.7,alpha=0.7,label=r'$\
↪hat{\phi}_{\mathrm{bin}}$')
    ax.errorbar(blf[0],blf[3], ecolor='k', capsize=0, xerr=np.vstack((blf[1], blf[2])),
                yerr=np.vstack((blf[4], blf[5])), fmt='None', zorder=4,alpha=0.5)

    plt.ylabel(r'$\log_{10}( \phi(z,L) ~/~ {\rm Mpc}^{-3} ~ \Delta L^{-1} )$',
↪fontsize=18)
    ax.legend(fontsize=12,loc='upper right')
    plt.xlabel(r'$L$',fontsize=18)
    plottitle = r'$z={0:.2f}$'.format(z_plot)
    plt.title(plottitle, size=14, y=1.01)
    x1,x2=max((min(blf[0])-2), min(lum)), min((max(blf[0])+2), max(lum))
    y1,y2=min(blf[3])-2,max(blf[3])+1
    plt.xlim(x1,x2)
    plt.ylim(y1,y2)
    plt.show()
```

```
blf=lfmc.get_binLF(Lbin=0.3,zbin=[0.3,0.7],plot_fig=False)    # get the binning LFs
plot_lfs_mcmc(z_plot=0.5, blf=blf, lf_mcmc=(lums, phi_mcmc, lower_error, upper_error))
```



The following lines of code implement the small sample approximation KDE method:

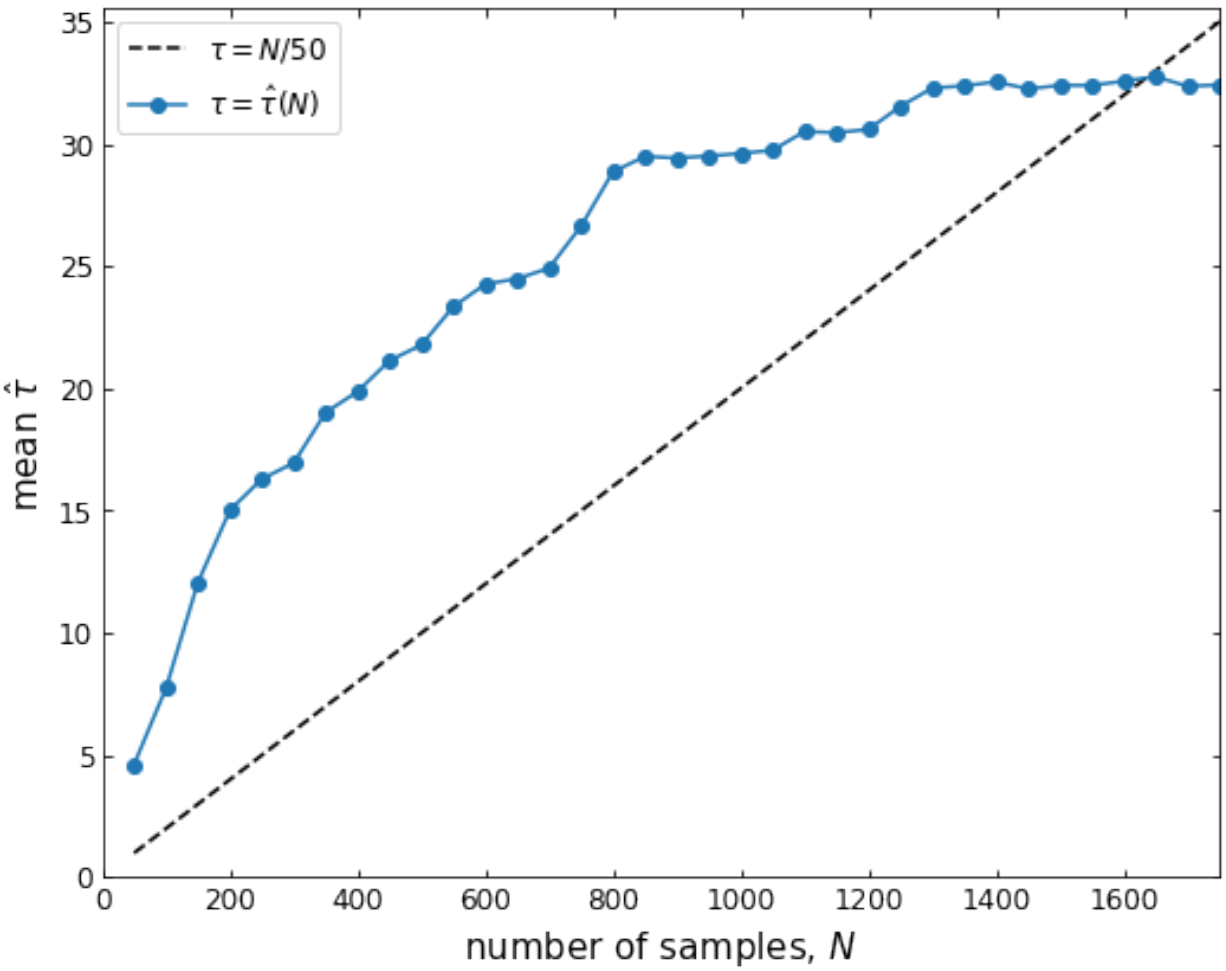
```
lf1a = kdeLF.KdeLF(sample_file='data.txt', solid_angle=omega, zbin=[3.5,4.5], f_lim=f_
↪lim,
                   H0=H0, Om0=Om0, small_sample=True, adaptive=True)
lf1a.get_optimal_h(quiet=True)
```

```
z & L data loaded
```

```
array([0.3916132, 1.       ])
```

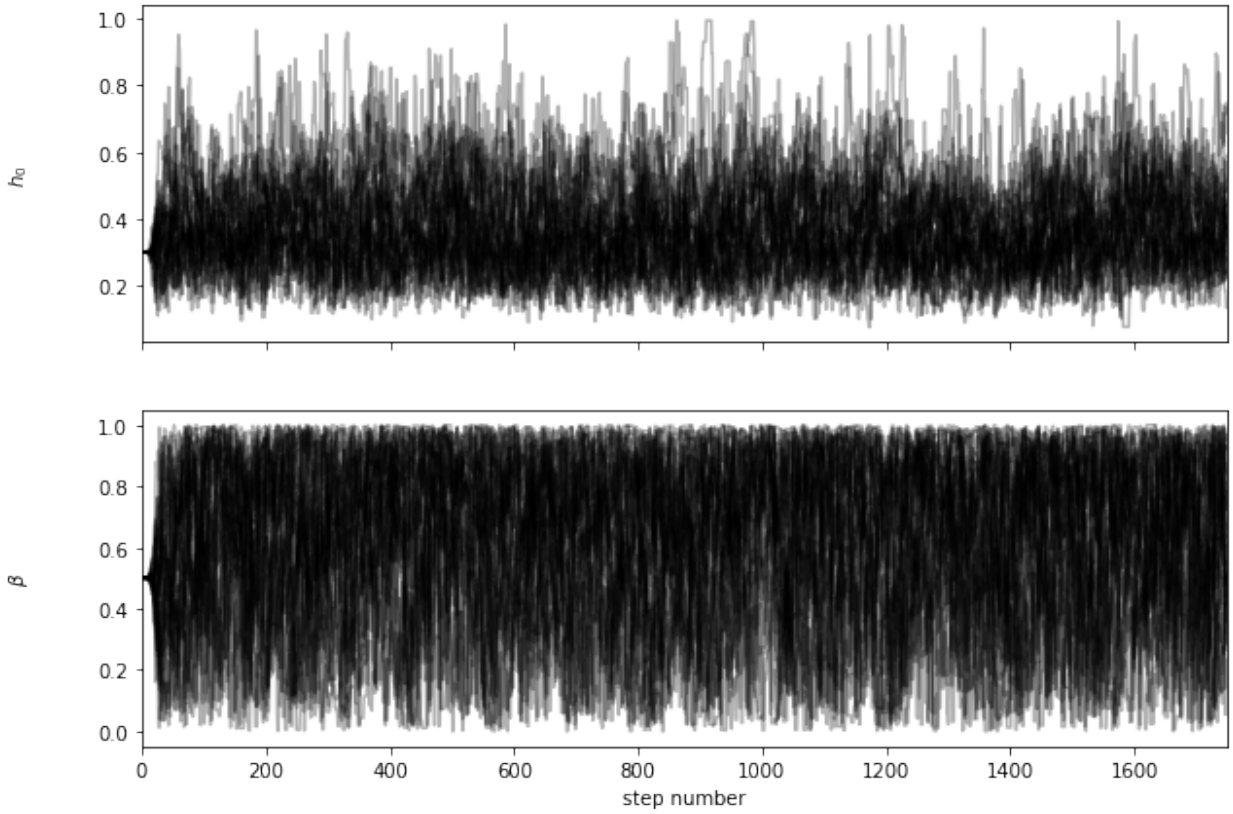
```
lf1a.run_mcmc(parallel=True, priors=[(0.01, 1.0),(0, 1)],initial_point=[0.39,0.5])
```

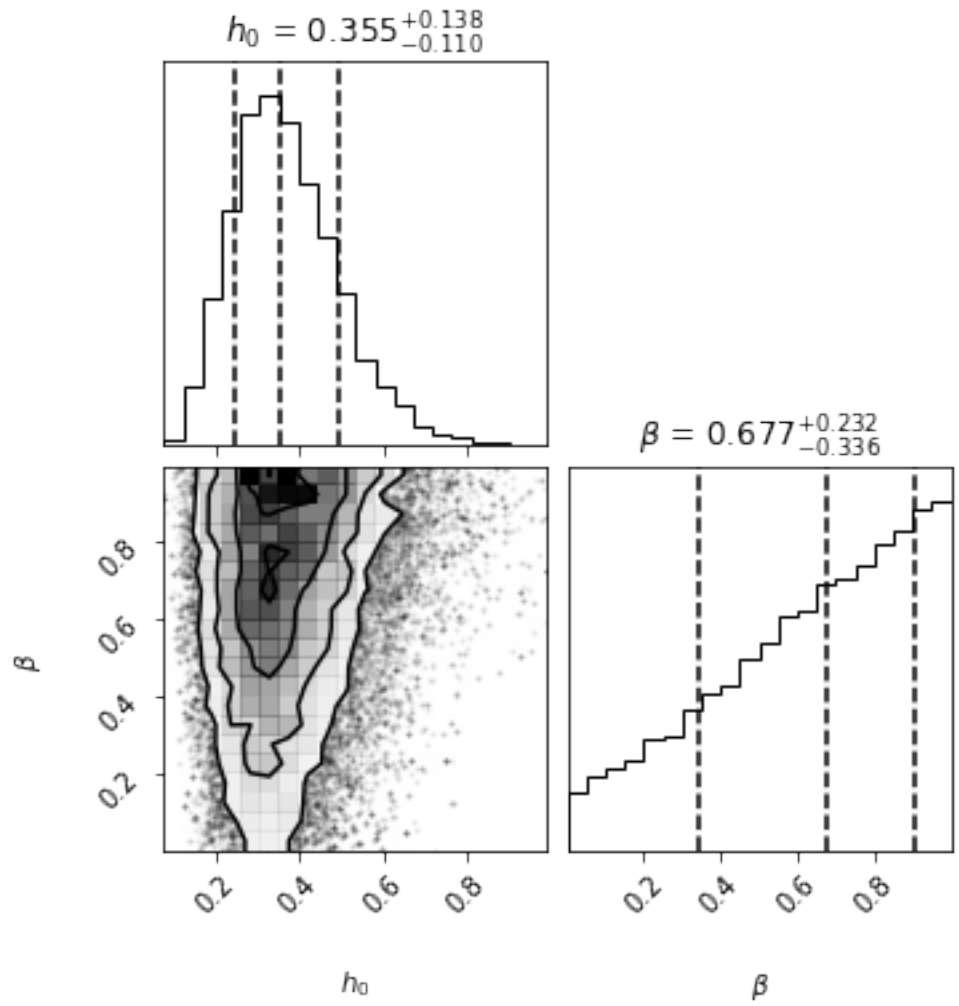
```
 58%|                | 1750/3000 [00:33<00:23, 52.30it/s]
```




```
lf1a.chain_analysis('chain_a3.5_4.5.h5')
```

```
burn-in: 67
thin: 15
flat chain shape: (3584, 2)
flat log prob shape: (3584,)

********** MCMC bestfit and 1 sigma errors **********
 h0 = 0.3553 - 0.1099 + 0.1384
beta = 0.6765 - 0.3360 + 0.2317
```

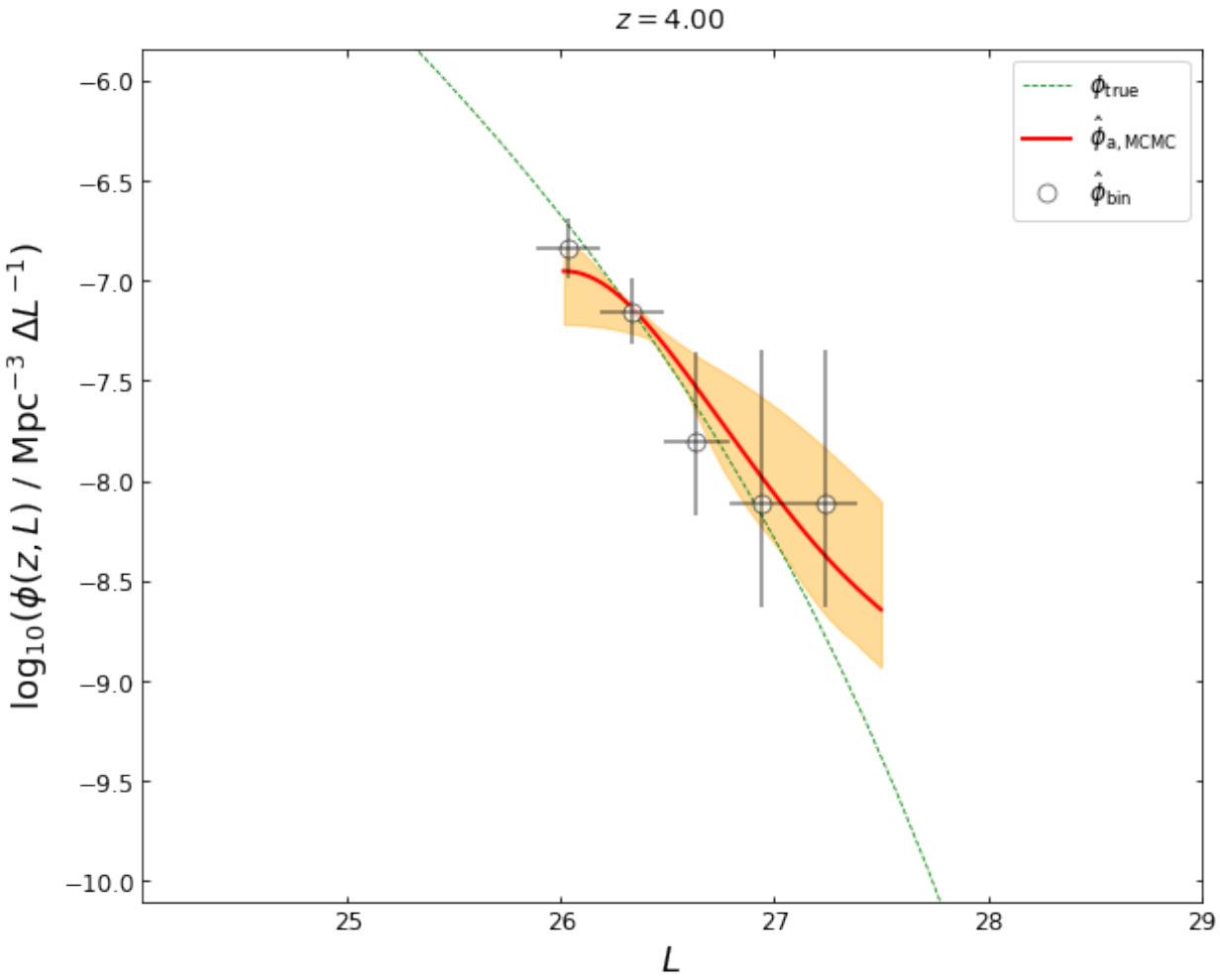

h_0
1.0
0.8
0.6
0.4
0.2
β
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
1200
1400
1600
step number




```
lf4_0 = lf1a.plot_posterior_LF(z=4.0,sigma=3,plot_fig=False,dpi=(50,3000))
lums, phi_mcmc, lower_error, upper_error = lf4_0
```

```
100%|| 3000/3000 [00:14<00:00, 208.11it/s]
```

```
blf=lf1a.get_binLF(Lbin=0.3,zbin=[3.5,4.5],plot_fig=False)    # get the binning LFs
plot_lfs_mcmc(z_plot=4.0, blf=blf, lf_mcmc=(lums, phi_mcmc, lower_error, upper_error))
```



2.5 Plot

```
%config InlineBackend.figure_format = "retina"

from matplotlib import rcParams

rcParams["savefig.dpi"] = 100
rcParams["figure.dpi"] = 100
rcParams["font.size"] = 20
```

coming soon …

2.6 Parallelization

```
%config InlineBackend.figure_format = "retina"

from matplotlib import rcParams

rcParams["savefig.dpi"] = 100
rcParams["figure.dpi"] = 100
rcParams["font.size"] = 20

import multiprocessing

multiprocessing.set_start_method("fork")
```

coming soon…

2.7 KS-test

```
%config InlineBackend.figure_format = "retina"

from matplotlib import rcParams

rcParams["savefig.dpi"] = 100
rcParams["figure.dpi"] = 100
rcParams["font.size"] = 20
```

coming soon…

2.8 MCMC

One important feature of our KDE method is involving the Bayesian inference. \texttt{run_mcmc} implements a fully Bayesian Markov Chain Monte Carlo (MCMC) method to determine the posterior distributions of bandwidths and other parameters (e.g., β for adaptive estimators). The MCMC core embedded in kdeLF is the Python package emcee (Foreman-Mackey et al.2013). The Bayesian method allows us to recover the parameters with a complete description of their uncertainties and degeneracies via calculating their probability density functions (PDFs). Then, by running the chain analysis subroutine, one can probe the shape of these PDFs, and the correlations among bandwidth parameters, giving more information than just the best-fit and the marginalized values for the parameters.

```
%config InlineBackend.figure_format = "retina"

from matplotlib import rcParams

rcParams["savefig.dpi"] = 100
rcParams["figure.dpi"] = 100
rcParams["font.size"] = 20
```

coming soon …

CHAPTER

THREE

LICENSE & ATTRIBUTION

CHAPTER

FOUR

CITATION

Please cite the following papers if you found this code useful in your research:

- Yuan, Z., Zhang, X., Wang, J., Cheng, X., & Wang, W. 2022, ApJS, 260, 10 (arXiv, ADS, BibTeX).
- Yuan, Z., Jarvis, M. J., & Wang, J. 2020, ApJS, 248, 1 (arXiv, ADS, BibTeX).

CHAPTER

FIVE

CONTRIBUTORS

- Wenjie Wang

CHAPTER

SIX

CHANGELOG

6.1 1.0.0 (2022-02-15)

- Initial release.