powerbox Documentation

Release 0.5.7

Steven Murray

Oct 24, 2018

Contents

Make arbitrarily structured, arbitrary-dimension boxes and log-normal mocks.

`powerbox` is a pure-python code for creating density grids (or boxes) that have an arbitrary two-point distribution (i.e. power spectrum). Primary motivations for creating the code were the simple creation of log-normal mock galaxy distributions, but the methodology can be used for other applications.

CHAPTER 1

Features

- Works in any number of dimensions.
- Really simple.
- Arbitrary isotropic power-spectra.
- Create Gaussian or Log-Normal fields
- Create discrete samples following the field, assuming it describes an over-density.
- Measure power spectra of output fields to ensure consistency.
- Seamlessly uses pyFFTW if available for ~double the speed.

CHAPTER 2

Installation

powerbox only depends on numpy >= 1.6.2, which will be installed automatically if powerbox is installed using pip (see below). Furthermore, it has the optional dependency of pyfftw, which if installed will offer ~2x performance increase in large fourier transforms. This will be seamlessly used if installed.

To install pyfftw, simply do:

```
pip install pyfftw
```

To install powerbox, do:

```
pip install powerbox
```

Alternatively, the bleeding-edge version from git can be installed with:

```
pip install git+git://github.com/steven-murray/powerbox.git
```

Finally, for a development installation, download the source code and then run (in the top-level directory):

```
pip install -e .
```

CHAPTER 3

Acknowledgment

If you find powerbox useful in your research, please cite the Journal of Open Source Software paper at https://doi.org/10.21105/joss.00850.

CHAPTER 4

QuickLinks

- Docs: https://powerbox.readthedocs.io
- Quickstart: http://powerbox.readthedocs.io/en/latest/demos/getting_started.html

CHAPTER 5

Contents

5.1 Examples

To help get you started using `powerbox`, we've compiled a few simple examples. Other examples can be found in the *API documentation* for each object or by looking at some of the tests.

5.1.1 Getting Started with Powerbox

This demo will get you started with using `powerbox` for the most common tasks.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np

        import powerbox as pbox
```

```
In [2]: pbox.__version__
```

```
Out[2]: '0.5.5'
```

Introduction

The power spectrum is ubiquitous in signal processing and spatial statistics. It measures the amplitude of fluctuations in a given field over a range of scales. Famously, for an homogeneous Gaussian random field (i.e. one in which the PDF of the value of the field over all locations yields a normal distribution), the power spectrum encodes the *entire* information of the field. It is thus a hugely useful tool in characterising close-to-homogeneous scalar fields with close-to-Gaussian PDFs. Examples of such fields are found in astrophysics, cosmology, fluid dynamics and various other scientific areas (often as idealised systems).

Mathematically, the power spectrum is merely the normalised absolute square of the Fourier Transform of a signal:

$$P(\vec{k}) \propto |\mathcal{F}(\vec{x})|^2. \tag{5.1}$$

In `powerbox`, we are concerned with fields of finite extent – usually subsamples of the true underlying field – that is, we are concerned with "boxes" which will be periodic representations of the underlying field. To obtain a power spectrum whose magnitude is independent of the volume of the box itself, we normalise by volume:

$$P(\vec{k}) = \frac{|\mathcal{F}(\vec{x})|^2}{V}, \tag{5.2}$$

yielding units of $[x]^n$ for the power, with n the number of dimensions in the box. In `powerbox`, normalising by the volume is optional (and true by default). Note that many conventions exist for the fourier transform, each of them entirely valid. In `powerbox` we support all of these conventions (see the `Changing Fourier Conventions` example notebook for details), but the default is set to mimic those used for cosmological structure:

$$\mathcal{F}_k = \int \mathcal{F}_x e^{-i\vec{k}\cdot\vec{x}} d^n\vec{x} \tag{5.3}$$

$$\mathcal{F}_x = (2\pi)^{-n/2} \int \mathcal{F}_k e^{i\vec{k}\cdot\vec{x}} d^n\vec{k}. \tag{5.4}$$

(5.5)

Note that if the field is homogeneous, then it is also isotropic – i.e. the field looks (statistically) the same in every direction at every point. In this case, the power spectrum can be reduced to a one-dimensional quantity, $P(k)$, where k is the magnitude of the fluctuation scale. It is these homogeneous and isotropic fields that `powerbox` is designed to work with.

The aim of `powerbox` is to make it easy to do the following two things:

1. Given a box representing a scalar field, evaluate $P(k)$.
2. Given a function $P(k)$ representing an isotropic power spectrum, construct a realisation of a finite, periodic, discretised field, $\delta_x(\vec{x})$ in n dimensions whose power is $P(k)$.

The first point is relatively simple (though it is useful to be able to do it easily and fast). The second point is less trivial: in particular, it is not fully defined. The information in $P(k)$ is a complete representation of δ_x iff the PDF of δ_x is Gaussian, and δ_x is periodic. In `powerbox`, we will always assume that δ_x is periodic. However, it is not always usefule to assume that its PDF is Gaussian. In particular, many scalar fields are positive-bounded (for example, density fields), while a Gaussian has support over all $\mathbb{R}$. This can easily yield un-physical boxes. In `powerbox`, we offer support for creating fields for two kinds of PDFs: the Gaussian and the log-normal (which is positive bounded).

Properties of the power spectrum

In `powerbox`, when creating scalar fields from a given power spectrum, the power spectrum itself is passed as a callable function of one parameter (i.e. k). In general, to be physically meaningful, the power spectrum should have the following properties:

1. $P(k) > 0, \forall k$
2. $\int_0^\infty P(k)dk$ converges.

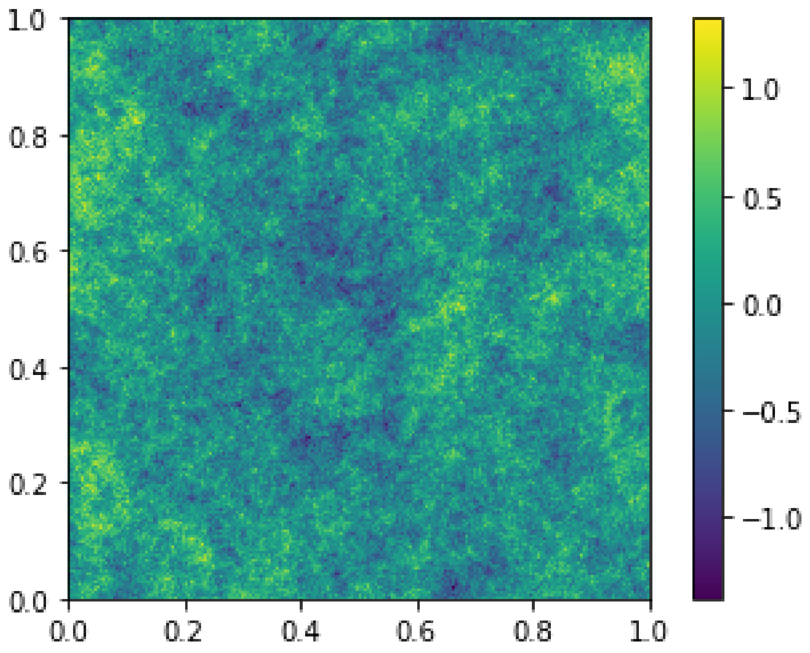
The first property is internally checked by `powerbox` and raises an error if not satisfied. The second property is not enforced, as in `powerbox` only a finite range of k is used (determined by box size and resolution). As long as the power is finite over this range, the field will be well-specified. Note that this assumes that the power outside of the range of k specified is zero.

Create a 2D Gaussian field with power-law power-spectrum

There are two useful classes in `powerbox`: the basic `PowerBox`, and one for log-normal fields: `LogNormalPowerBox`. To see their options just use `help(pbox.PowerBox)`.

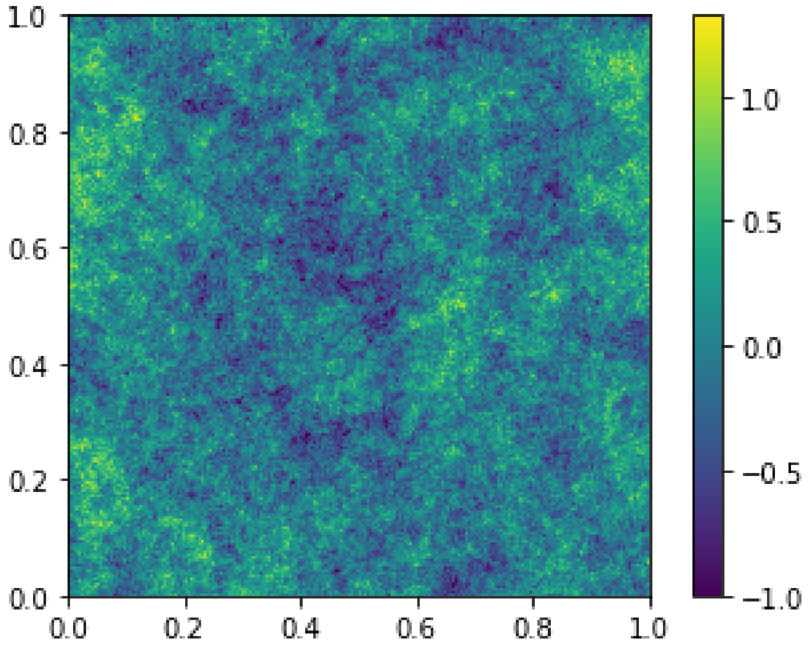
For a basic 2D Gaussian field with a power-law power-spectrum, one can use the following:

```
In [3]: pb = pbox.PowerBox(
            N=512,                     # Number of grid-points in the box
            dim=2,                     # 2D box
            pk = lambda k: 0.1*k**-2., # The power-spectrum
            boxlength = 1.0,           # Size of the box (sets the units of k in pk)
            seed = 1010                # Set a seed to ensure the box looks the same every time (opt
        )

        plt.imshow(pb.delta_x(),extent=(0,1,0,1))
        plt.colorbar()
        plt.show()
```



The `delta_x` output is *always* zero-mean, so it can be interpreted as an over-density field, $\rho(x)/\bar{\rho} - 1$. The caveat to this is that an overdensity field is physically invalid below -1. To ensure the physical validity of the field, the option `ensure_physical` can be set, which clips the field:

```
In [4]: pb = pbox.PowerBox(
            N=512,                     # Number of grid-points in the box
            dim=2,                     # 2D box
            pk = lambda k: 0.1*k**-2., # The power-spectrum
            boxlength = 1.0,           # Size of the box (sets the units of k in pk)
            seed = 1010,               # Set a seed to ensure the box looks the same every time (opt
            ensure_physical=True       # ** Ensure the delta_x is a physically valid over-density **
        )


        plt.imshow(pb.delta_x(),extent=(0,1,0,1))
        plt.colorbar()
        plt.show()
```

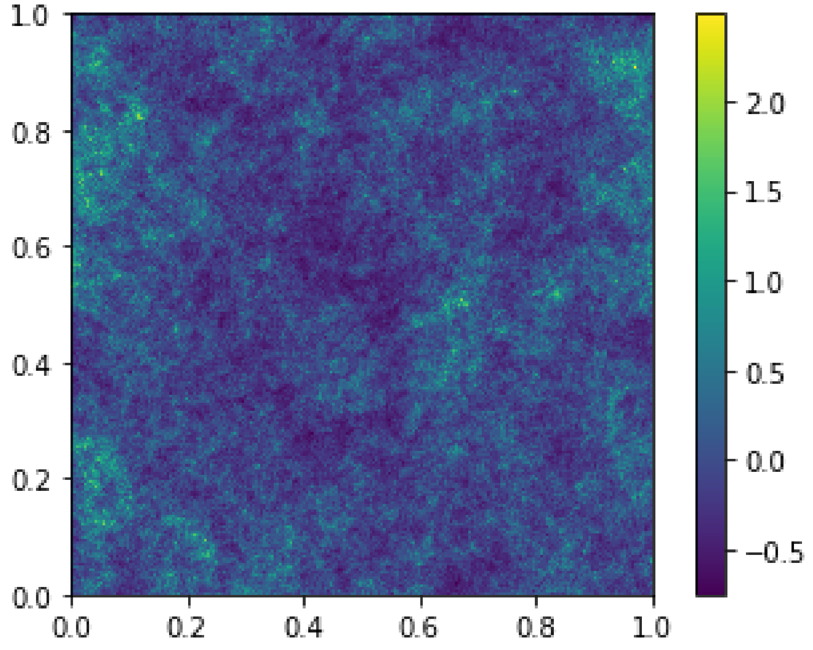


If you are actually dealing with over-densities, then this clipping solution is probably a bit hacky. What you want is a log-normal field…

Create a 2D Log-Normal field with power-law power spectrum

The `LogNormalPowerBox` class is called in exactly the same way, but the resulting field has a log-normal pdf with the same power spectrum.

```
In [5]: lnpb = pbox.LogNormalPowerBox(
            N=512,                        # Number of grid-points in the box
            dim=2,                        # 2D box
            pk = lambda k: 0.1*k**-2., # The power-spectrum
            boxlength = 1.0,              # Size of the box (sets the units of k in pk)
            seed = 1010                   # Use the same seed as our powerbox
        )
        plt.imshow(lnpb.delta_x(),extent=(0,1,0,1))
        plt.colorbar()
        plt.show()
```

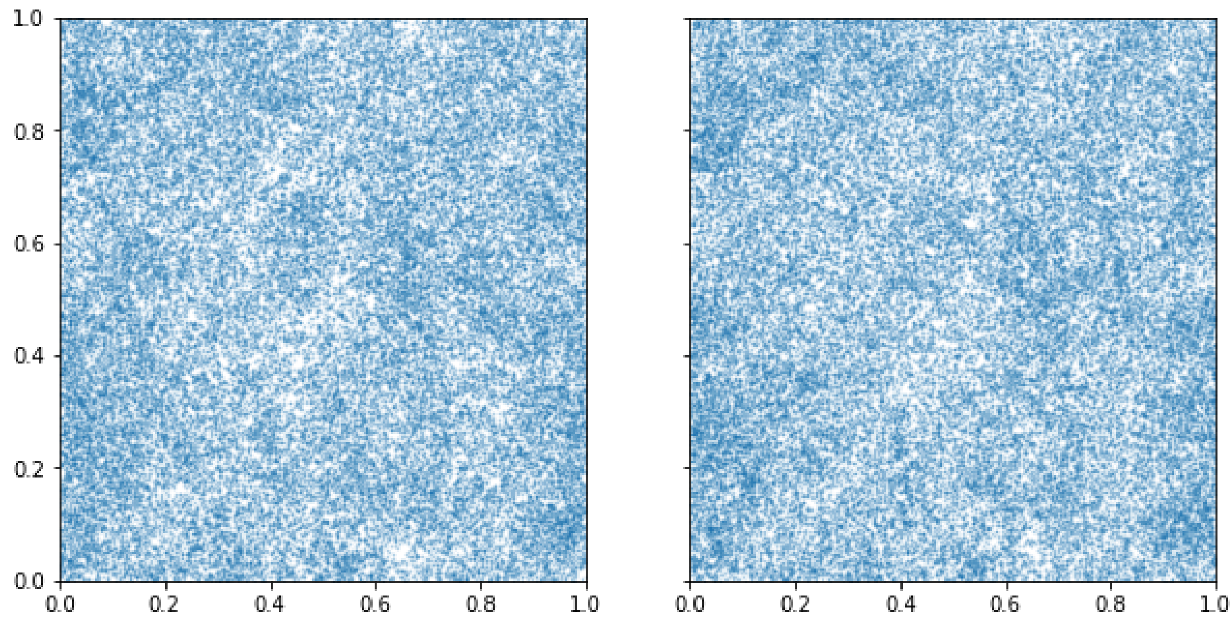


Again, the `delta_x` is zero-mean, but has a longer positive tail due to the log-normal nature of the distribution. This means it is always greater than -1, so that the over-density field is always physical.

Create some discrete samples on the field

`powerbox` lets you easily create samples that follow the field:

```
In [6]: fig, ax = plt.subplots(1,2, sharex=True,sharey=True,gridspec_kw={"hspace":0}, subplot_kw={"yl

        # Create a discrete sample using the PowerBox instance.
        samples = pb.create_discrete_sample(nbar=50000,      # nbar specifies the number density
                                            min_at_zero=True  # by default the samples are centred a
                                           )
        ln_samples = lnpb.create_discrete_sample(nbar=50000, min_at_zero=True)

        # Plot the samples
        ax[0].scatter(samples[:,0],samples[:,1], alpha=0.2,s=1)
        ax[1].scatter(ln_samples[:,0],ln_samples[:,1],alpha=0.2,s=1)
        plt.show()
```



Within each grid-cell, the placement of the samples is uniformly random. The samples can instead be placed on the cell edge by setting `randomise_in_cell` to `False`.

Check the power-spectrum of the field

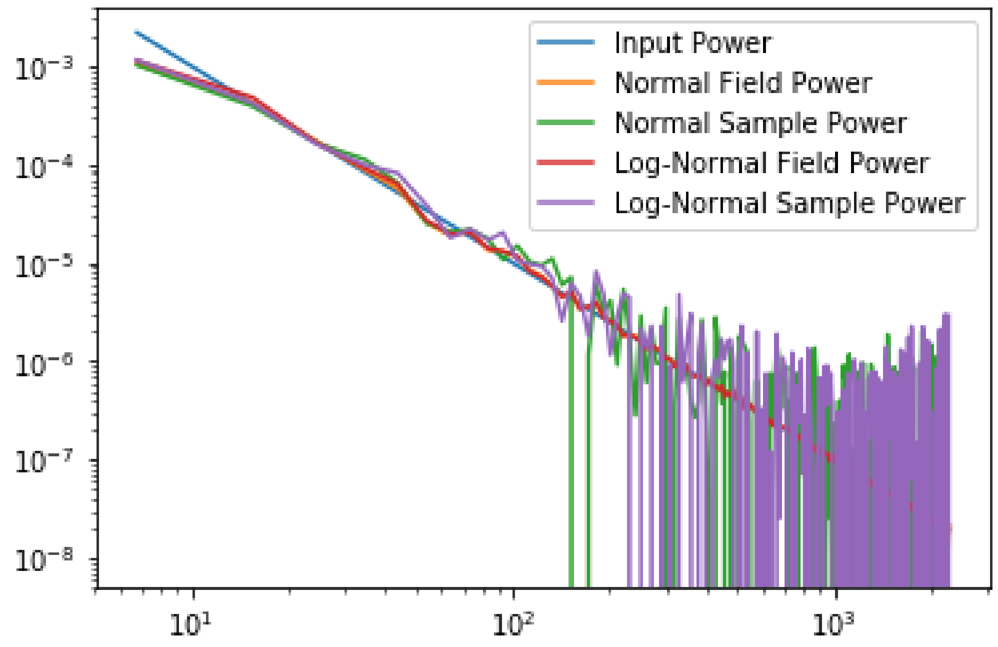
`powerbox` also contains a function for computing the (isotropic) power-spectrum of a field. This function accepts either a box defining the field values at every co-ordinate, *or* a set of discrete samples. In the latter case, the routine returns the power spectrum of over-densities, which matches the field that produced them. Let's go ahead and compute the power spectrum of our boxes, both from the samples and from the fields themselves:

```
In [7]: from powerbox import get_power
```

```
In [8]: # Only two arguments required when passing a field
        p_k_field, bins_field = get_power(pb.delta_x(), pb.boxlength)
        p_k_lnfield, bins_lnfield = get_power(lnpb.delta_x(), lnpb.boxlength)

        # The number of grid points are also required when passing the samples
        p_k_samples, bins_samples = get_power(samples, pb.boxlength,N=pb.N)
        p_k_lnsamples, bins_lnsamples = get_power(ln_samples, lnpb.boxlength,N=lnpb.N)
```

Now we can plot them all together to ensure they line up:

```
In [9]: plt.plot(bins_field, 0.1*bins_field**-2., label="Input Power")

        plt.plot(bins_field, p_k_field,label="Normal Field Power")
        plt.plot(bins_samples, p_k_samples,label="Normal Sample Power")
        plt.plot(bins_lnfield, p_k_lnfield,label="Log-Normal Field Power")
        plt.plot(bins_lnsamples, p_k_lnsamples,label="Log-Normal Sample Power")

        plt.legend()
        plt.xscale('log')
        plt.yscale('log')
```



5.1.2 How Does Powerbox Work?

It may be useful to understand the workings of powerbox to some extent – either to diagnose performance issues or to understand its behaviour in certain contexts.

The basic algorithm (for a Gaussian field) is the following:

1. Given a box length L (parameter `boxlength`) and number of cells along a side, N (parameter `N`), as well as Fourier convention parameters (a, b), determine wavenumbers along a side of the box: $k = 2\pi j/(bL)$, for $j \in (-N/2, ..., N/2)$.
2. From these wavenumbers along each side, determine the *magnitude* of the wavenumbers at every point of the d-dimensional box, $k_j = \sqrt{\sum_{i=1}^{d} k_{i,j}^2}$.
3. Create an array, G_j, which assigns a complex number to each grid point. The complex number will have magnitude drawn from a standard normal, and phase distributed evenly on $(0, 2\pi)$.
4. Determine $\delta_{k,j} = G_j \sqrt{P(k_j)}$.
5. Determine $\delta_x = V\mathcal{F}^{-1}(\delta_k)$, with $V = \prod_{i=1}^{d} L_i$.

For a Log-Normal field, the steps are slightly more complex, and involve determining the power spectrum that *would be* required on a Gaussian field to yield the same power spectrum for a log-normal field. The details of this approach can be found in Coles and Jones (1991) or Beutler et al. (2011).

One characteristic of this algorithm is that it contains *no information* below the resolution scale L/N. Thus, a good rule-of-thumb is to choose N large enough to ensure that the smallest scale of interest is covered by a factor of 1.5, i.e., if the smallest length-scale of interest is s, then use $N = 1.5L/s$.

The range of k used with this choice of N also depends on the Fourier Convention used. For the default convention of $b = 1$, the smallest scales are equivalent to $k = \pi N/L$.

5.1.3 Create a log-normal mock dark-matter distribution

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np

        import powerbox as pbox
        pbox.__version__
```

```
Out[1]: '0.5.5'
```

In this demo, we create a mock dark-matter distribution, based on the cosmological power spectrum. To generate the power-spectrum we use the `hmf` code (https://github.com/steven-murray/hmf).

```
In [2]: import hmf
        hmf.__version__
```

```
Out[2]: '3.0.3'
```

The box can be set up like this:

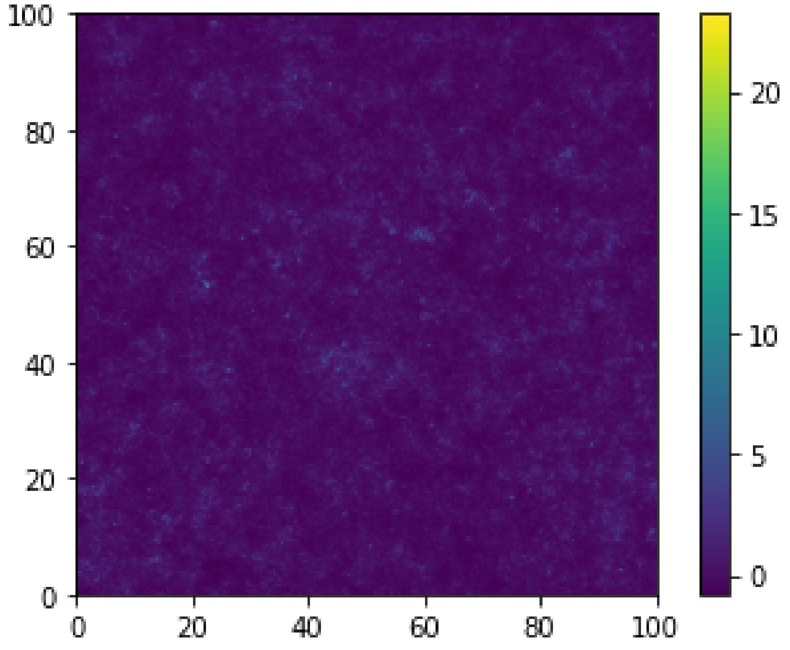
```
In [3]: from scipy.interpolate import InterpolatedUnivariateSpline as spline
        import numpy as np

        # Set up a MassFunction instance to access its cosmological power-spectrum
        mf = hmf.MassFunction(z=0)

        # Generate a callable function that returns the cosmological power spectrum.
        spl = spline(np.log(mf.k),np.log(mf.power),k=2)
        power = lambda k : np.exp(spl(np.log(k)))

        # Create the power-box instance. The boxlength is in inverse units of the k of which pk is a
        # Mpc/h in this case.
        pb = pbox.LogNormalPowerBox(N=256, dim=3, pk = power, boxlength= 500., seed=1234)
```

We will be using the `delta_x` quantity a couple of times throughout this demo, so we save it to a variable here. Otherwise, `powerbox` will recalculate it each time it is called.

```
In [4]: deltax = pb.delta_x()
```

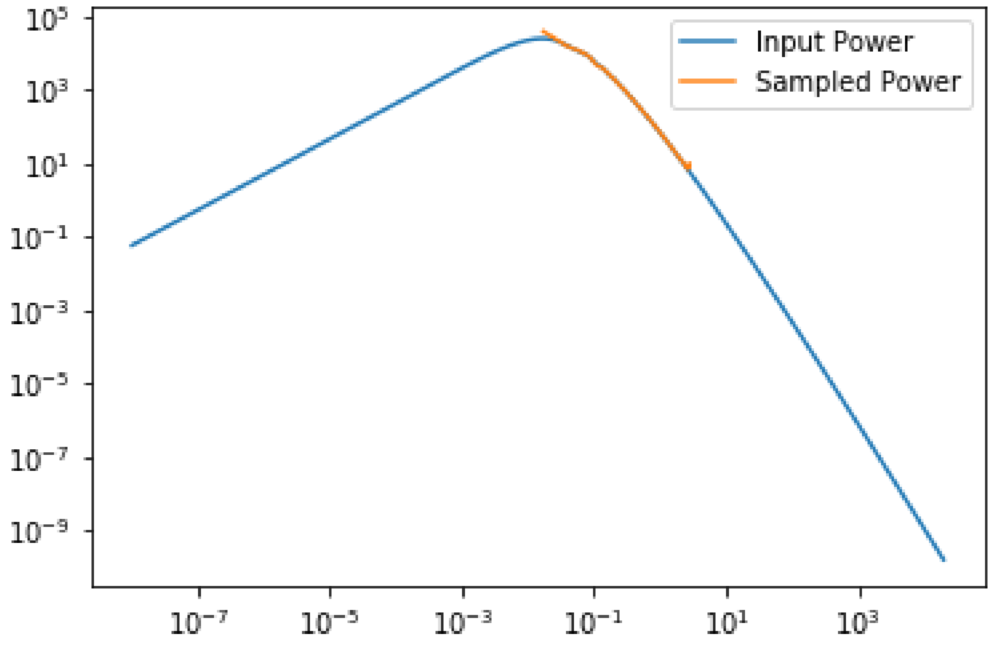
Now we can make a plot of a slice of the density field:

```
In [5]: plt.imshow(np.mean(deltax[:100,:,:],axis=0),extent=(0,100,0,100))
        plt.colorbar()
        plt.show()
```



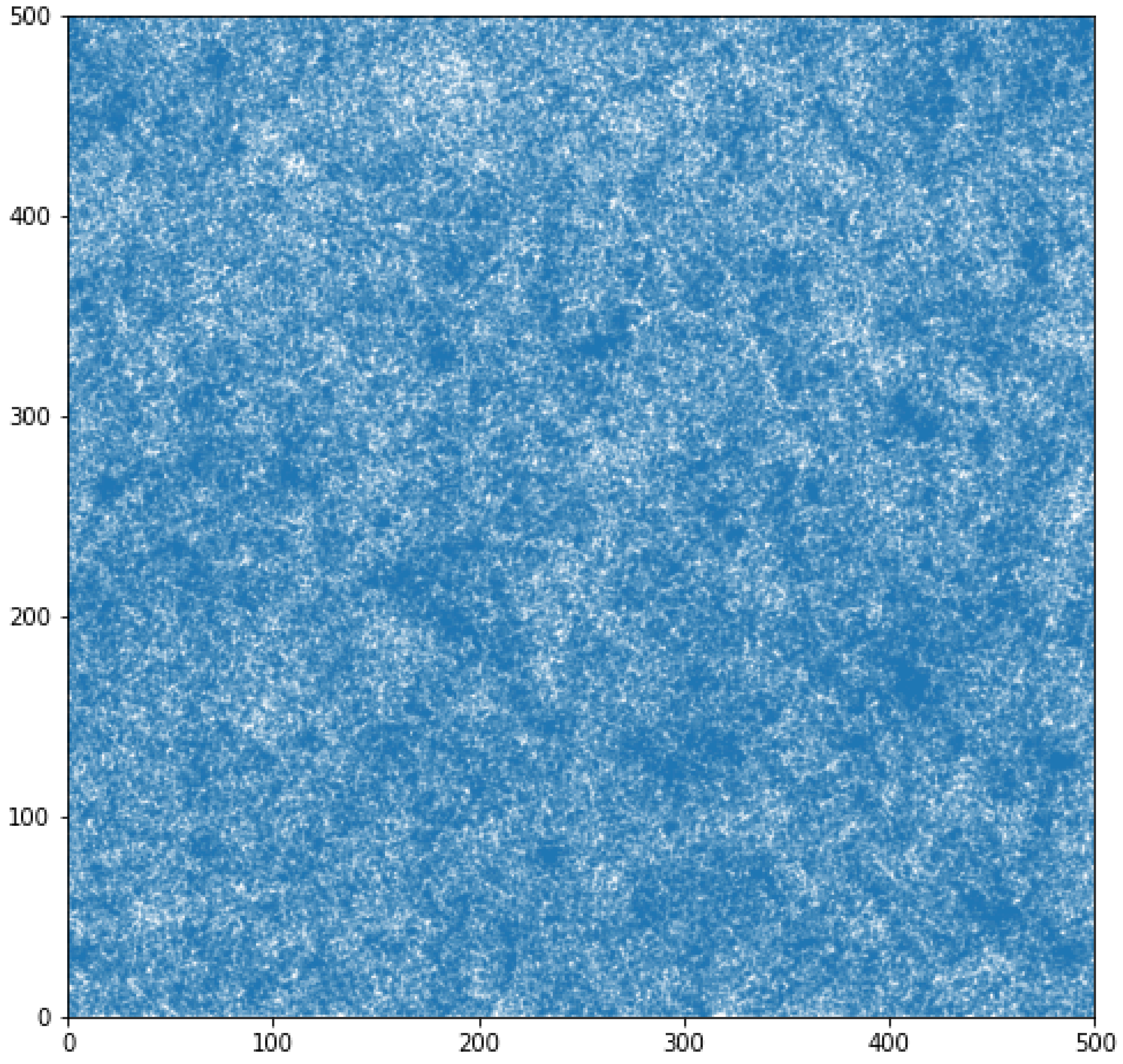
And we can also compare the power-spectrum of the output field to the input power:

```
In [6]: p_k, kbins = pbox.get_power(deltax,pb.boxlength)
        plt.plot(mf.k,mf.power,label="Input Power")
        plt.plot(kbins,p_k,label="Sampled Power")
        plt.xscale('log')
        plt.yscale('log')
        plt.legend()
        plt.show()
```



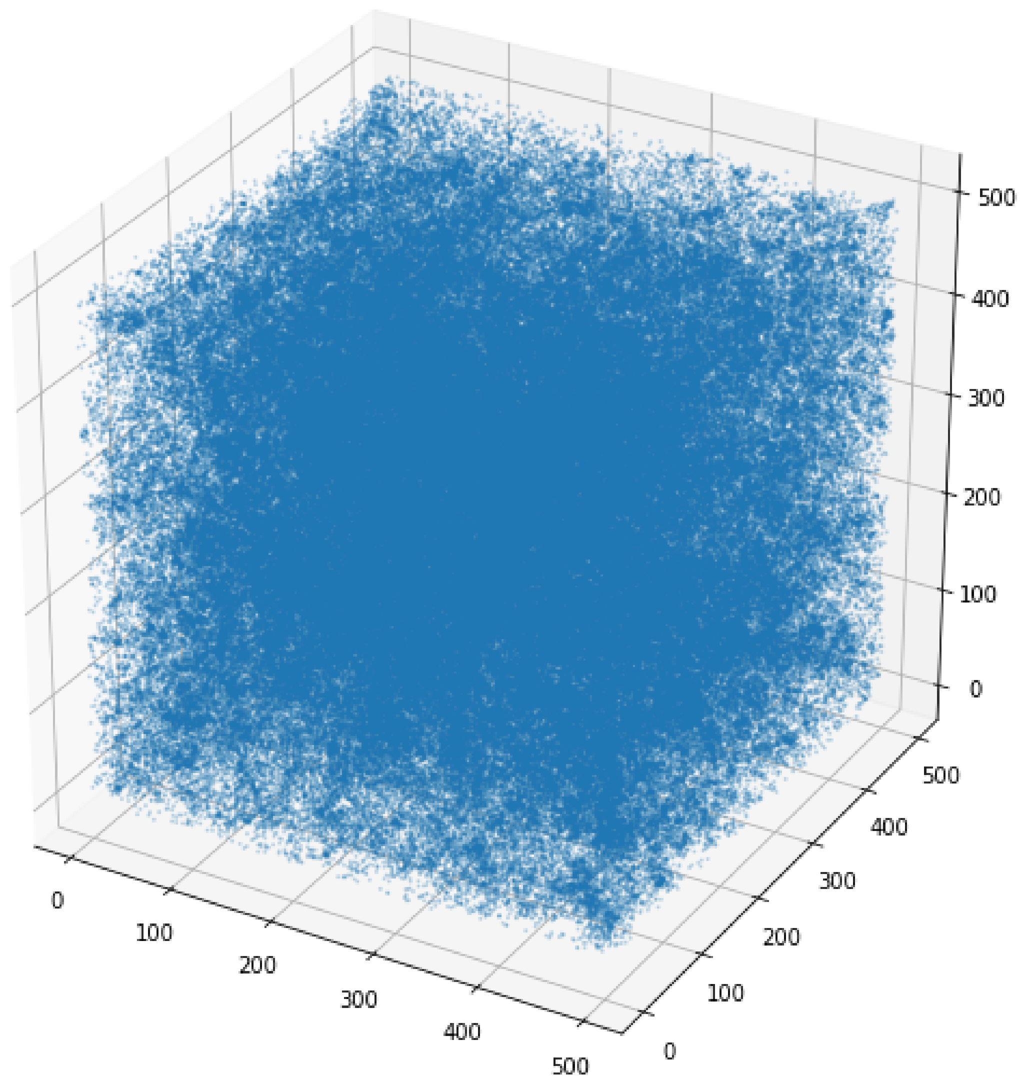
Furthermore, we can sample a set of discrete particles on the field and plot them:

```
In [7]: particles = pb.create_discrete_sample(nbar=0.003,min_at_zero=True)

        plt.figure(figsize=(8,8))
        plt.scatter(particles[:,0],particles[:,1],s=np.sqrt(100./particles[:,2]),alpha=0.2)
        plt.xlim(0,500)
        plt.ylim(0,500)
        plt.show()
```



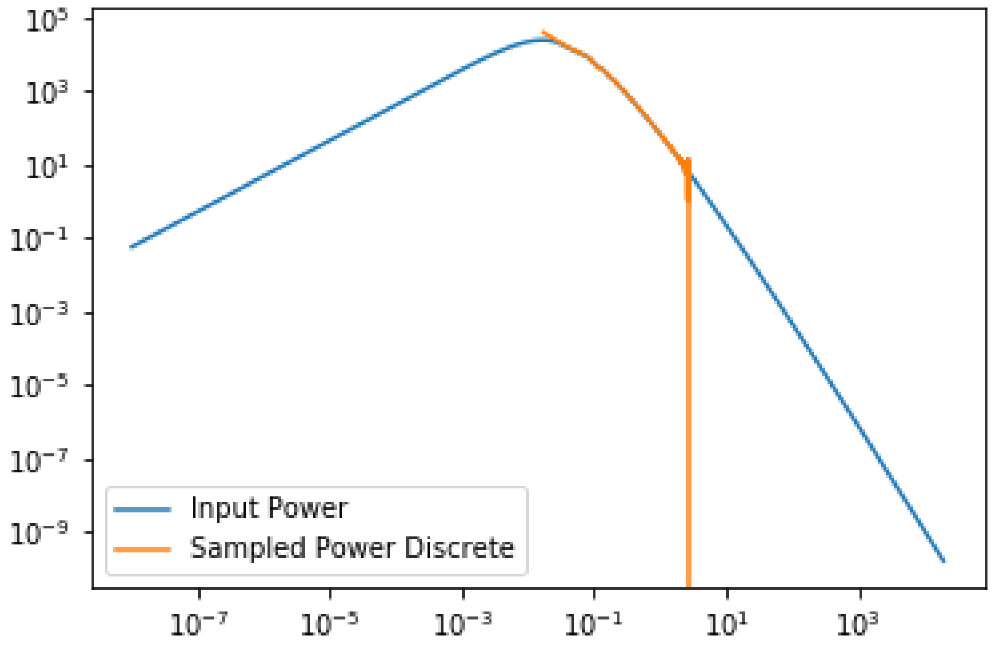
Or plot them in 3D!

```
In [8]: from mpl_toolkits.mplot3d import Axes3D

        fig = plt.figure(figsize=(10,10))
        ax = fig.add_subplot(111, projection='3d')

        ax.scatter(particles[:,0], particles[:,1], particles[:,2],s=1,alpha=0.2)
        plt.show()
```



Then check that the power-spectrum of the sample matches the input:

```
In [9]: p_k_sample, kbins_sample = pbox.get_power(particles, pb.boxlength,N=pb.N)

        plt.plot(mf.k,mf.power,label="Input Power")
        plt.plot(kbins_sample,p_k_sample,label="Sampled Power Discrete")
        plt.xscale('log')
        plt.yscale('log')
        plt.legend()
        plt.show()
```



5.1.4 Changing Fourier Conventions

The `powerbox` package allows for arbitrary Fourier conventions. Since (continuous) Fourier Transforms can be defined using different definitions of the frequency term, and varying normalisations, we allow any consistent combination of these, using the same parameterisation that Mathematica uses, i.e.:

$$F(k) = \left(\frac{|b|}{(2\pi)^{1-a}}\right)^{n/2} \int f(r) e^{-ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{r}$$

for the forward-transform and

$$f(r) = \left(\frac{|b|}{(2\pi)^{1+a}}\right)^{n/2} \int F(k) e^{+ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{k}$$

for its inverse. Here n is the number of dimensions in the transform, and a and b are free to be any real number. Within `powerbox`, b is taken to be positive.

The most common choice of parameters is $(a, b) = (0, 2\pi)$, which are the parameters that for example `numpy` uses. In cosmology (a field which `powerbox` was written in the context of), a more usual choice is $(a, b) = (1, 1)$, and so these are the defaults within the `PowerBox` classes.

In this notebook we provide some examples on how to deal with these conventions.

Doing the DFT right.

In many fields, we are concerned primarily with the *continuous* FT, as defined above. However, typically we must evaluate this numerically, and therefore use a DFT or FFT. While the conversion between the two is simple, it is all too easy to forget which factors to normalise by to get back the analogue of the continuous transform.

That's why in `powerbox` we provide some fast fourier transform functions that do all the hard work for you. They not only normalise everything correctly to produce the continuous transform, they also return the associated fourier-dual co-ordinates. And they do all this for arbitrary conventions, as defined above. And they work for any number of dimensions.

Let's take a look at an example, using a simple Gaussian field in 2D:

$$f(x) = e^{-\pi r^2},$$

where $r^2 = x^2 + y^2$.

The Fourier transform of this field, using the standard mathematical convention is:

$$\int e^{-\pi r^2} e^{-2\pi i k \cdot x} d^2 x = e^{-\pi k^2},$$

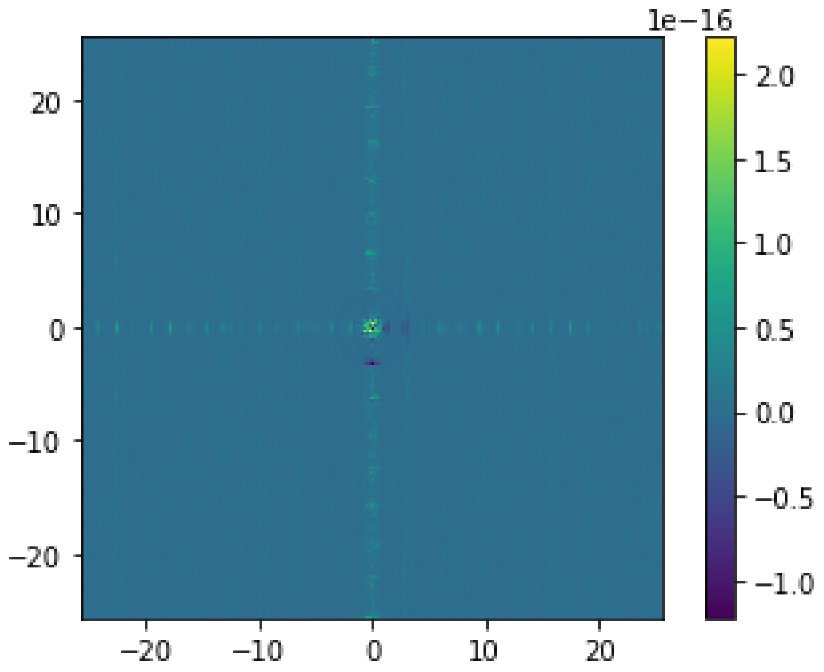
where $k^2 = k_x^2 + k_y^2$.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np

        from powerbox import fft,ifft
        from powerbox.powerbox import _magnitude_grid
        import powerbox as pbox
```

```
In [2]: pbox.__version__
```

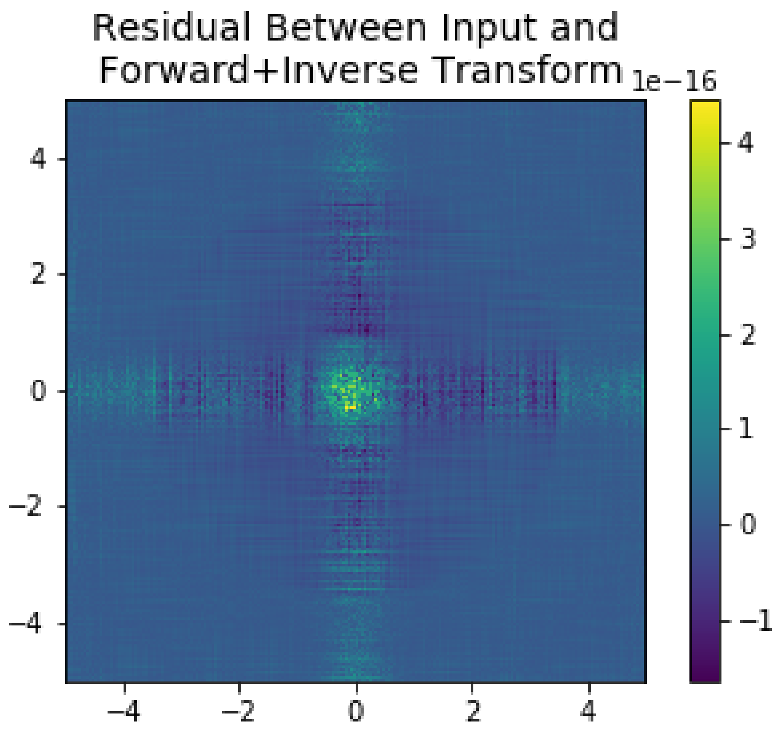
```
Out[2]: '0.5.5'
```

```
In [3]: # Parameters of the field
        L = 10.
        N = 512
        dx = L/N

        x = np.arange(-L/2,L/2,dx)[:N] # The 1D field grid
        r = _magnitude_grid(x,dim=2)   # The magnitude of the co-ordinates on a 2D grid
        field = np.exp(-np.pi*r**2)    # Create the field

        # Generate the k-space field, the 1D k-space grid, and the 2D magnitude grid.
        k_field, k, rk = fft(field,L=L,          # Pass the field to transform, and its size
                             ret_cubegrid=True   # Tell it to return the grid of magnitudes.
                            )

        # Plot the field minus the analytic result
        plt.imshow(np.abs(k_field)-np.exp(-np.pi*rk**2),extent=(k.min(),k.max(),k.min(),k.max()))
        plt.colorbar();
```



We can now of course do the inverse transform, to ensure that we return the original:

```
In [4]: x_field, x_, rx = ifft(k_field, L = L,   # Note we can pass L=L, or Lk as the extent of the
                              ret_cubegrid=True)

        plt.imshow(np.abs(x_field)-field,extent=(x.min(),x.max(),x.min(),x.max()))
        plt.title("Residual Between Input and\n Forward+Inverse Transform", fontsize=14)
        plt.colorbar()
        plt.show();
```



We can also check that the xgrid returned is the same as the input xgrid:

```
In [5]: x_ -x
```

```
Out[5]: array([[0., 0., 0., ..., 0., 0., 0.],
               [0., 0., 0., ..., 0., 0., 0.]])
```

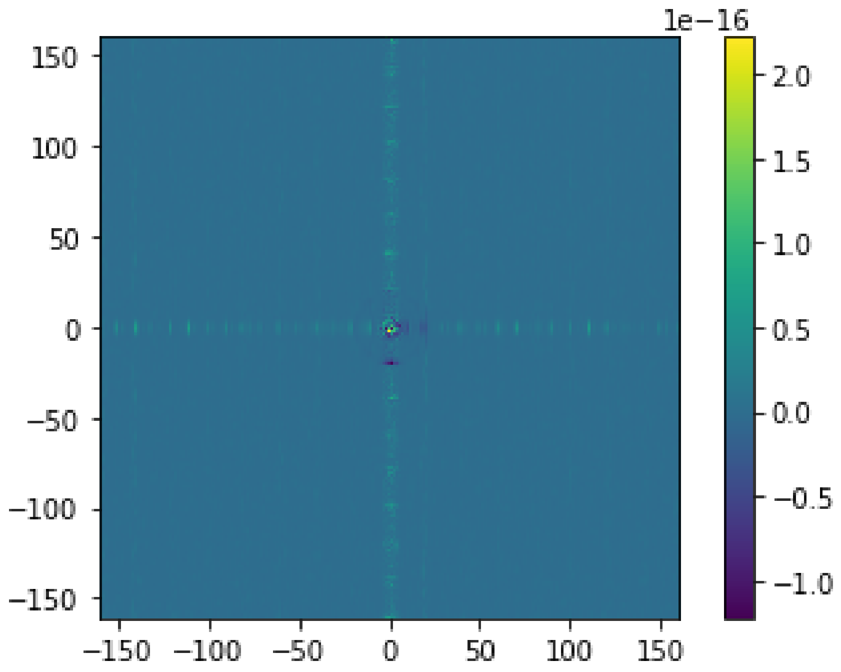
Changing the convention

Suppose we instead required the transform

$$\int e^{-\pi r^2} e^{-i\nu\cdot x} d^2x = e^{-\nu^2/4\pi}.$$

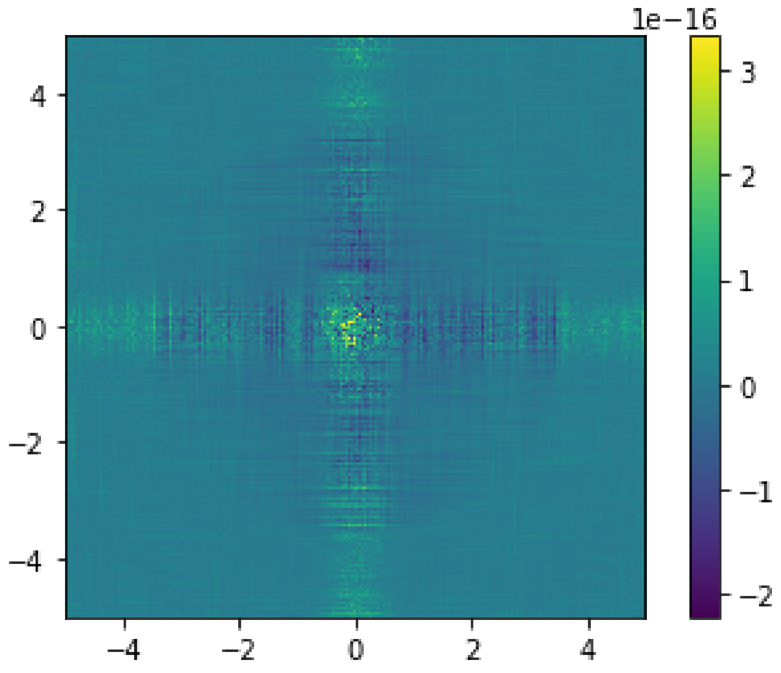
This is the same transform but with the Fourier-convention $(a,b) = (1,1)$. We would do this like:

```
In [6]: # Generate the k-space field, the 1D k-space grid, and the 2D magnitude grid.
        k_field, k, rk = fft(field,L=L,           # Pass the field to transform, and its size
                             ret_cubegrid=True,    # Tell it to return the grid of magnitudes.
                             a=1,b=1               # SET THE FOURIER CONVENTION
                            )

        # Plot the field minus the analytic result
        plt.imshow(np.abs(k_field)-np.exp(-1./(4*np.pi)*rk**2),extent=(k.min(),k.max(),k.min(),k.max
        plt.colorbar();
```



Again, specifying the inverse transform with these conventions gives back the original:

```
In [7]: x_field, x_, rx = ifft(k_field, L = L,    # Note we can pass L=L, or Lk as the extent of the 
                                 ret_cubegrid=True,
                                 a=1,b=1
                                )

        plt.imshow(np.abs(x_field)-field,extent=(x.min(),x.max(),x.min(),x.max()))
        plt.colorbar()
        plt.show();
```



Mixing up conventions

It may be that sometimes the forward and inverse transforms in a certain problem will have different conventions. Say the forward transform has parameters (a, b), and the inverse has parameters (a', b'). Then first taking the forward transform, and *then* inverting it (in n-dimensions) would yield:

$$\left(\frac{b'}{b(2\pi)^{a'-a}}\right)^{n/2} f\left(\frac{b'r}{b}\right),$$

and doing it the other way would yield:

$$\left(\frac{b}{b'(2\pi)^{a'-a}}\right)^{n/2} F\left(\frac{bk}{b'}\right).$$

The `fft` and `ifft` functions handle these easily. For example, if $(a, b) = (0, 2\pi)$ and $(a', b') = (0, 1)$, then the 2D forward-then-inverse transform should be

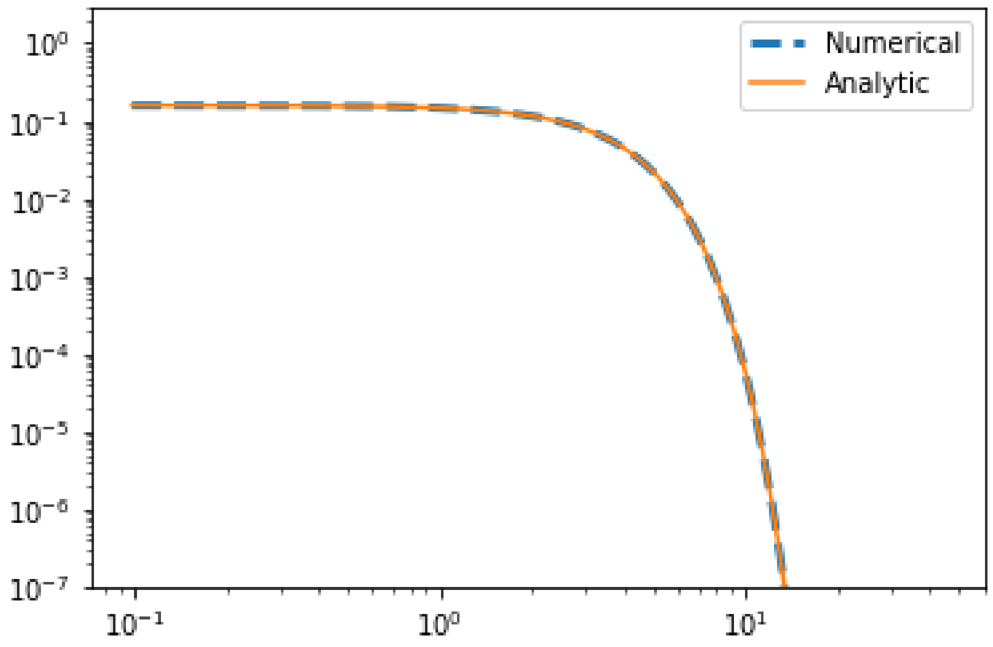
$$f(r/(2\pi))/2\pi,$$

and the inverse-then-forward should be

$$2\pi F(2\pi k).$$

```
In [8]: # Do the forward transform
        k_field,k,rk = fft(field,L=L,a=0,b=2*np.pi, ret_cubegrid=True)

        # Do the inverse transform, ensuring the boxsize is correct
        mod_field,modx,modr = ifft(k_field,Lk=-2*k.min(),a=0,b=1, ret_cubegrid=True)

        mod_field, bins = pbox.angular_average(mod_field, modr, 300)

        plt.plot(bins,mod_field, label="Numerical",lw=3,ls='--')
        plt.plot(bins,np.exp(-np.pi*(bins/(2*np.pi))**2)/(2*np.pi),label="Analytic")
        plt.legend()
        plt.yscale('log')
        plt.xscale('log')
        plt.ylim(1e-7,3)
        plt.show()
```

```
/home/steven/miniconda3/envs/powerbox/lib/python3.6/site-packages/numpy/core/numeric.py:492: ComplexW
  return array(a, dtype, copy=False, order=order)
```



Using Different Conventions in Powerbox

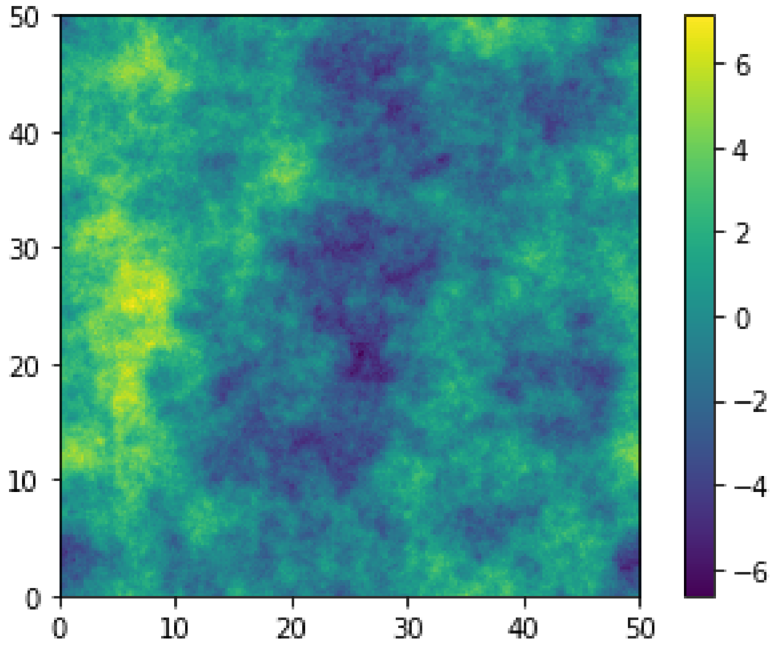
These fourier-transform wrappers are used inside powerbox to do the heavy lifting. That means that one can pass a power spectrum which has been defined with arbitrary conventions, and receive a fully consistent box back.

Let's say, for example, that the fourier convention in your field was to use $(a, b) = (0, 1)$, so that the power spectrum of a 2D field, δ_x was given by

$$P(k) = \frac{1}{2\pi} \int \delta_x e^{-ikx} d^2x.$$

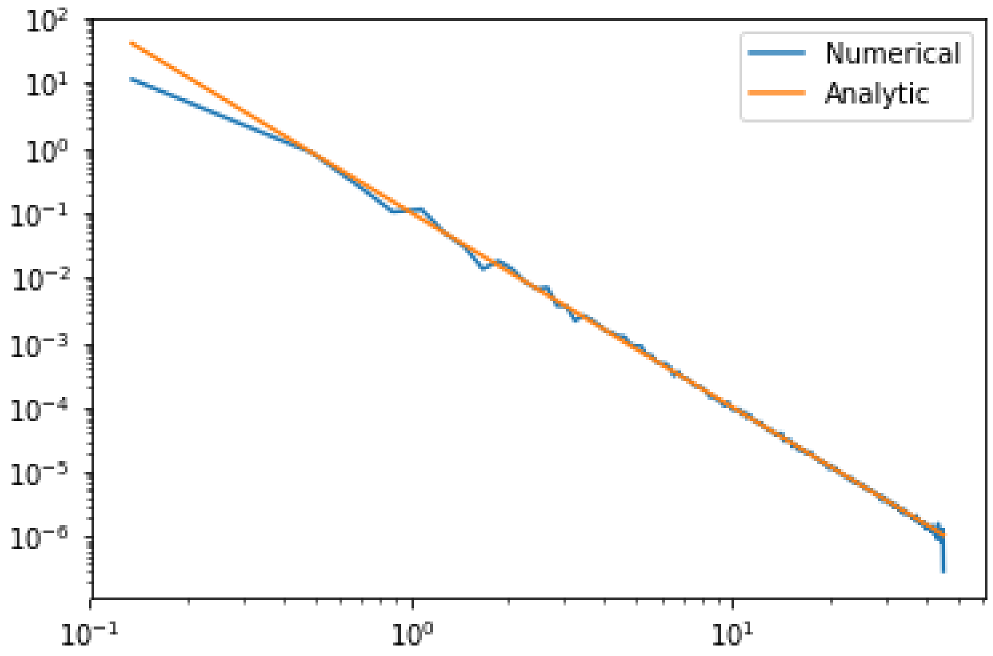
We now wish to create a realisation with a power spectrum following these conventions. Let's say the power spectrum is $P(k) = 0.1k^{-2}$.

```
In [9]: pb = pbox.PowerBox(
            N=512,dim=2,pk = lambda k : 0.1*k**-3.,
            a=0, b=1,          # Set the Fourier convention
            boxlength=50.0     # Has units of inverse k
        )

        plt.imshow(pb.delta_x(),extent=(0,50,0,50))
        plt.colorbar()
        plt.show()
```



When we check the power spectrum, we also have to remember to set the Fourier convention accordingly:

```
In [10]: power, kbins = pbox.get_power(pb.delta_x(), pb.boxlength, a= 0,b =1)

         plt.plot(kbins,power,label="Numerical")
         plt.plot(kbins,0.1*kbins**-3.,label="Analytic")
         plt.legend()
         plt.xscale('log')
         plt.yscale('log')
         plt.show()
```



5.2 License

5.3 Changelog

5.3.1 v0.5.7 [24 Oct 2018]

Enhancements

- Added ability to use weights on k-modes in *get_power*.

Bugfixes

- Fixed bug on using *ignore_zero_mode* introduced in v0.5.6
- Added tests for *ignore_zero_mode* and *k_weights*

5.3.2 v0.5.6 [23 Oct 2018]

Enhancements

- Added *ignore_zero_mode* parameter to *get_power*.

Bugfixes

- Removed redundant *seed* parameter from *create_discrete_sample()*.

5.3.3 v0.5.5 [19 July 2018]

Bugfixes

- log_bins wasn't being passed through to angular_average correctly.

Enhancements

- `angular_average()` no longer requires coords to be passed as box of magnitudes.
- improved docs.
- fixed source divide by zero warning in PowerBox()

5.3.4 v0.5.4 [30 May 2018]

Enhancements

- Added ability to do angular averaging in log-space bins
- When not all radial bins have co-ordinates in them, a more reasonable warning message is emitted.
- Removed redundant bincount call when only summing, not averaging (angularly).

Bugfixes

- Now properly deals with co-ordinates outside the bin range in angular_average (will only make a difference when bins is passed as a vector). Note that this has meant that by default the highest-valued co-ordinate in the box will *not* contribute to any bins any more.
- Fixed a bunch of tests in test_power which were using the wrong power index!

Internals

- Re-factored getting radial bins into _getbins() function.

5.3.5 v0.5.3 [22 May 2018]

Bugfixes

- Fixed a bug introduced in v0.5.1 where using bin_ave=False in angular_average_nd would fail.

5.3.6 v0.5.2 [17 May 2018]

Enhancements

- Added ability to calculate the variance of an angularly averaged quantity.
- Removed a redundant calculation of the bin weights in angular_average

Internals

- Updated version numbers of dev requirements.

5.3.7 v0.5.1 [4 May 2018]

Enhancements

- Added ability to *not* have dimensionless power spectra from get_power.
- Also return linearly-spaced radial bin edges from angular_average_nd
- Python 3 compatibility

Bugfixes

- Fixed bug where field was modified in-place unexpectedly in angular_average
- Now correctly flattens weights before getting the field average in angular_average_nd

v0.5.0 [7 Nov 2017] ———————~ **Features**

- Input boxes to get_power no longer need to have same length on every dimension.
- New angular_average_nd function to average over first n dimensions of an array.

Enhancements

- Huge (5x or so) speed-up for angular_average function (with resulting speedup for get_power).
- Huge memory reduction in fft/ifft routines, with potential loss of some speed (TODO: optimise)
- Better memory consumption in PowerBox classes, at the expense of an API change (cached properties no longer cached, or properties).
- Modified fftshift in dft to handle astropy Quantity objects (bit of a hack really)

Bugfixes

- Fixed issue where if the boxlength was passed as an integer (to fft/ifft), then incorrect results occurred.
- Fixed issue where incorrect first_edge assignment in get_power resulted in bad power spectrum. No longer require this arg.

5.3.8 v0.4.3 [29 March 2017]

Bugfixes

- Fixed volume normalisation in get_power.

5.3.9 v0.4.2 [28 March 2017]

Features

- Added ability to cross-correlate boxes in get_power.

5.3.10 v0.4.1

Bugfixes

- Fixed cubegrid return value for dft functions when input boxes have different sizes on each dimension.

5.3.11 v0.4.0

Features

- Added fft/ifft wrappers which consistently return fourier transforms with arbitrary Fourier conventions.
- Boxes now may be composed with arbitrary Fourier conventions.
- Documentation!

Enhancements

- New test to compare LogNormalPowerBox with standard PowerBox.
- New project structure to make for easier location of functions.
- Code quality improvements
- New tests, better coverage.

Bugfixes

- Fixed incorrect boxsize for an odd number of cells
- Ensure mean density is correct in LogNormalPowerBox

5.3.12 v0.3.2

Bugfixes

- Fixed bug in pyFFTW cache setting

5.3.13 v0.3.1

Enhancements

- New interface with pyFFTW to make fourier transforms ~twice as fast. No difference to the API.

5.3.14 v0.3.0

Features

- New functionality in *get_power* function to measure power-spectra of discrete samples.

Enhancements

- Added option to not store discrete positions in class (just return them)
- *get_power* now more streamlined and intuitive in its API

5.3.15 v0.2.3 [11 Jan 2017]

Enhancements

- Improved estimation of power (in `get_power`) for lowest k bin.

5.3.16 v0.2.2 [11 Jan 2017]

Bugfixes

- Fixed a bug in which the output power spectrum was a factor of sqrt(2) off in normalisation

5.3.17 v0.2.1 [10 Jan 2017]

Bugfixes

- Fixed output of `create_discrete_sample` when not randomising positions.

Enhancements

- New option to set bounds of discrete particles to (0, boxlength) rather than centring at 0.

5.3.18 v0.2.0 [10 Jan 2017]

Features

- New `LogNormalPowerBox` class for creating log-normal fields

Enhancements

- Restructuring of code for more flexibility after creation. Now requires `cached_property` package.

5.3.19 v0.1.0 [27 Oct 2016]

First working version. Only Gaussian fields working.

5.4 Authors

- Steven Murray

5.4.1 Comments, corrections and suggestions

- Chris Jordan

5.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.5.2 Documentation improvements

powerbox could always use more documentation, whether as part of the official powerbox docs, in docstrings, or even on the web in blog posts, articles, and such.

5.5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/steven-murray/powerbox/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.5.4 Development

To set up *powerbox* for local development:

1. Fork powerbox (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/powerbox.git
```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

   ```
   tox
   ```

5. Commit your changes and push your branch to GitHub:

   ```
   git add .
   git commit -m "Your detailed description of your changes."
   git push origin name-of-your-bugfix-or-feature
   ```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `CONTRIBUTORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

5.6 API Summary

5.6.1 powerbox.powerbox Module

A module defining two classes which can create arbitrary-dimensional fields with given power spectra. One such function produces *Gaussian* fields, and the other *LogNormal* fields.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though . . .

In principle, these may be extended to other 1-point density distributions by subclassing *PowerBox* and over-writing the same methods as are over-written in *LogNormalPowerBox*.

Classes

LogNormalPowerBox(*args, **kwargs)	Calculate Log-Normal density fields with given power spectra.
PowerBox(N, pk[, dim, boxlength, ...])	Calculate real- and fourier-space Gaussian fields generated with a given power spectrum.

LogNormalPowerBox

class powerbox.powerbox.**LogNormalPowerBox**(**args*, ***kwargs*)

Bases: *powerbox.powerbox.PowerBox*

Calculate Log-Normal density fields with given power spectra.

See the documentation of *PowerBox* for a detailed explanation of the arguments, as this class has exactly the same arguments.

This class calculates an (over-)density field of arbitrary dimension given an input isotropic power spectrum. In this case, the field has a log-normal distribution of over-densities, always yielding a physically valid field.

Examples

To create a log-normal over-density field:

```
>>> from powerbox import LogNormalPowerBox
>>> lnpb = LogNormalPowerBox(100,lambda k : k**-7./5.,dim=2, boxlength=1.0)
>>> overdensities = lnpb.delta_x
>>> grid = lnpb.x
>>> radii = lnpb.r
```

To plot the overdensities:

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(pb.delta_x)
```

Compare the fields from a Gaussian and Lognormal realisation with the same power:

```
>>> lnpb = LogNormalPowerBox(300,lambda k : k**-7./5.,dim=2, boxlength=1.0)
>>> pb = PowerBox(300,lambda k : k**-7./5.,dim=2, boxlength=1.0)
>>> fig,ax = plt.subplots(2,1,sharex=True,sharey=True,figsize=(12,5))
>>> ax[0].imshow(lnpb.delta_x,aspect="equal",vmin=-1,vmax=lnpb.delta_x.max())
>>> ax[1].imshow(pb.delta_x,aspect="equal",vmin=-1,vmax = lnpb.delta_x.max())
```

To create and plot a discrete version of the field:

```
>>> positions = lnpb.create_discrete_sample(nbar=1000.0, # Number density in␣
↪terms of boxlength units
>>>                                         randomise_in_cell=True)
>>> plt.scatter(positions[:,0],positions[:,1],s=2,alpha=0.5,lw=0)
```

Attributes Summary

kvec	The vector of wavenumbers along a side
r	The radial position of every point in the grid
x	The co-ordinates of the grid along a side

Methods Summary

correlation_array()	The correlation function from the input power, on the grid
create_discrete_sample(nbar[, . . .])	Assuming that the real-space signal represents an over-density with respect to some mean, create a sample of tracers of the underlying density distribution.
delta_k()	A realisation of the delta_k, i.e.
delta_x()	The real-space over-density field, from the input power spectrum
gauss_hermitian()	A random array which has Gaussian magnitudes and Hermitian symmetry
gaussian_correlation_array()	The correlation function required for a Gaussian field to produce the input power on a lognormal field
gaussian_power_array()	The power spectrum required for a Gaussian field to produce the input power on a lognormal field
k()	The entire grid of wavenumber magitudes
power_array()	The Power Spectrum (volume normalised) at *self.k*

Attributes Documentation

kvec
The vector of wavenumbers along a side

r
The radial position of every point in the grid

x
The co-ordinates of the grid along a side

Methods Documentation

correlation_array()
The correlation function from the input power, on the grid

create_discrete_sample(*nbar*, *randomise_in_cell=True*, *min_at_zero=False*, *store_pos=False*)
Assuming that the real-space signal represents an over-density with respect to some mean, create a sample of tracers of the underlying density distribution.

Parameters

nbar [float] Mean tracer density within the box.

randomise_in_cell [bool, optional] Whether to randomise the positions of the tracers within the cells, or put them at the grid-points (more efficient).

min_at_zero [bool, optional] Whether to make the lower corner of the box at the origin, otherwise the centre of the box is at the origin.

store_pos [bool, optional] Whether to store the sample of tracers as an instance variable *tracer_positions*.

Returns

tracer_positions [float, array_like] `(n, d)`-array, with `n` the number of tracers and `d` the number of dimensions. Each row represents a single tracer's co-ordinates.

`delta_k``()`
A realisation of the delta_k, i.e. the gaussianised square root of the unitless power spectrum (i.e. the Fourier co-efficients)

`delta_x``()`
The real-space over-density field, from the input power spectrum

`gauss_hermitian``()`
A random array which has Gaussian magnitudes and Hermitian symmetry

`gaussian_correlation_array``()`
The correlation function required for a Gaussian field to produce the input power on a lognormal field

`gaussian_power_array``()`
The power spectrum required for a Gaussian field to produce the input power on a lognormal field

`k``()`
The entire grid of wavenumber magitudes

`power_array``()`
The Power Spectrum (volume normalised) at *self.k*

PowerBox

class `powerbox.powerbox.`**`PowerBox`**(*N*, *pk*, *dim=2*, *boxlength=1.0*, *ensure_physical=False*, *a=1.0*, *b=1.0*, *vol_normalised_power=True*, *seed=None*)
Bases: `object`

Calculate real- and fourier-space Gaussian fields generated with a given power spectrum.

Parameters

N [int] Number of grid-points on a side for the resulting box (equivalently, number of wavenumbers to use).

pk [callable] A callable of a single (vector) variable *k*, which is the isotropic power spectrum. The relationship of the *k* of which this is a function to the real-space co-ordinates, *x*, is determined by the parameters `a,b`.

dim [int, default 2] Number of dimensions of resulting box.

boxlength [float, default 1.0] Length of the final signal on a side. This may have arbitrary units, so long as *pk* is a function of a variable which has the inverse units.

ensure_physical [bool, optional] Interpreting the power spectrum as a spectrum of density fluctuations, the minimum physical value of the real-space field, *`delta_x()`*, is -1. With `ensure_physical` set to `True`, *`delta_x()`* is clipped to return values >-1. If this is happening a lot, consider using *`LogNormalPowerBox`*.

a,b [float, optional] These define the Fourier convention used. See `powerbox.dft` for details. The defaults define the standard usage in *cosmology* (for example, as defined in Cosmological Physics, Peacock, 1999, pg. 496.). Standard numerical usage (eg. numpy) is (a,b) = (0,2pi).

vol_normalised_power [bool, optional] Whether the input power spectrum, `pk`, is volume-weighted. Default True because of standard cosmological usage.

seed: int, optional A random seed to define the initial conditions. If not set, it will remain random, and each call to eg. `delta_x()` will produce a *different* realisation.

Notes

A number of conventions need to be listed.

The conventions of using *x* for "real-space" and *k* for "fourier space" arise from cosmology, but this does not affect anything – *x* could just as well stand for "time domain" and *k* for "frequency domain".

The important convention is the relationship between *x* and *k*, or in other words, whether *k* is interpreted as an angular frequency or ordinary frequency. By default, because of cosmological conventions, *k* is an angular frequency, so that the fourier transform integrand is delta_k*exp(-ikx). The conventions can be changed arbitrarily by setting the `a,b` parameters (see `powerbox.dft` for details).

The primary quantity of interest is `delta_x()`, which is a zero-mean Gaussian field with a power spectrum equivalent to that which was input. Being zero-mean enables its direct interpretation as an overdensity field, and this interpretation is enforced in the `make_discrete_sample()` method.

Note: None of the n-dimensional arrays that are created within the class are stored, due to the inefficiency in memory consumption that this would imply. Thus, each large array is created and *returned* by their respective method, to be stored/discarded by the user.

Warning: Due to the above note, repeated calls to eg. `delta_x()` will produce *different* realisations of the real-space field, unless the *seed* parameter is set in the constructor.

Examples

To create a 3-dimensional box of gaussian over-densities, gridded into 100 bins, with cosmological conventions, and a power-law power spectrum, simply use

```
>>> pb = PowerBox(100,lambda k : 0.1*k**-3., dim=3, boxlength=100.0)
>>> overdensities = pb.delta_x()
>>> grid = pb.x
>>> radii = pb.r
```

To create a 2D turbulence structure, with arbitrary units, once can use

```
>>> import matplotlib.pyplot as plt
>>> pb = PowerBox(1000, lambda k : k**-7./5.)
>>> plt.imshow(pb.delta_x())
```

Attributes Summary

`kvec`	The vector of wavenumbers along a side
`r`	The radial position of every point in the grid
`x`	The co-ordinates of the grid along a side

Methods Summary

`create_discrete_sample`(nbar[, . . .])	Assuming that the real-space signal represents an over-density with respect to some mean, create a sample of tracers of the underlying density distribution.
`delta_k`()	A realisation of the delta_k, i.e.
`delta_x`()	The realised field in real-space from the input power spectrum
`gauss_hermitian`()	A random array which has Gaussian magnitudes and Hermitian symmetry
`k`()	The entire grid of wavenumber magitudes
`power_array`()	The Power Spectrum (volume normalised) at *self.k*

Attributes Documentation

kvec
The vector of wavenumbers along a side

r
The radial position of every point in the grid

x
The co-ordinates of the grid along a side

Methods Documentation

create_discrete_sample(*nbar*, *randomise_in_cell=True*, *min_at_zero=False*, *store_pos=False*)
Assuming that the real-space signal represents an over-density with respect to some mean, create a sample of tracers of the underlying density distribution.

Parameters

nbar [float] Mean tracer density within the box.

randomise_in_cell [bool, optional] Whether to randomise the positions of the tracers within the cells, or put them at the grid-points (more efficient).

min_at_zero [bool, optional] Whether to make the lower corner of the box at the origin, otherwise the centre of the box is at the origin.

store_pos [bool, optional] Whether to store the sample of tracers as an instance variable *tracer_positions*.

Returns

tracer_positions [float, array_like] `(n, d)`-array, with `n` the number of tracers and `d` the number of dimensions. Each row represents a single tracer's co-ordinates.

delta_k()
A realisation of the delta_k, i.e. the gaussianised square root of the power spectrum (i.e. the Fourier co-efficients)

delta_x()
The realised field in real-space from the input power spectrum

gauss_hermitian()
A random array which has Gaussian magnitudes and Hermitian symmetry

k()
The entire grid of wavenumber magitudes

power_array()
The Power Spectrum (volume normalised) at *self.k*

5.6.2 powerbox.dft Module

A module defining some "nicer" fourier transform functions.

We define only two functions – an arbitrary-dimension forward transform, and its inverse. In each case, the transform is designed to replicate the continuous transform. That is, the transform is volume-normalised and obeys correct Fourier conventions.

The actual FFT backend is provided by `pyFFTW` if it is installed, which provides a significant speedup, and multi-threading.

Conveniently, we allow for arbitrary Fourier convention, according to the scheme in http://mathworld.wolfram.com/FourierTransform.html. That is, we define the forward and inverse n-dimensional transforms respectively as

$$F(k) = \sqrt{\frac{|b|}{(2\pi)^{1-a}}}^n \int f(r) e^{-ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{r}$$

and

$$f(r) = \sqrt{\frac{|b|}{(2\pi)^{1+a}}}^n \int F(k) e^{+ib\mathbf{k}\cdot\mathbf{r}} d^n\mathbf{k}.$$

In both transforms, the corresponding co-ordinates are returned so a completely consistent transform is simple to get. This makes switching from standard frequency to angular frequency very simple.

We note that currently, only positive values for b are implemented (in fact, using negative b is consistent, but one must be careful that the frequencies returned are descending, rather than ascending).

Functions

fft(X[, L, Lk, a, b, axes, ret_cubegrid])	Arbitrary-dimension nice Fourier Transform.
ifft(X[, Lk, L, a, b, axes, ret_cubegrid])	Arbitrary-dimension nice inverse Fourier Transform.
fftfreq(N[, d, b])	Return the fourier frequencies for a box with N cells, using general Fourier convention.
fftshift(x, *args, **kwargs)	The same as numpy's fftshift, except that it preserves units (if Astropy quantities are used)
ifftshift(x, *args, **kwargs)	The same as numpy's ifftshift, except that it preserves units (if Astropy quantities are used)

fft

`powerbox.dft.fft`(*X*, *L=None*, *Lk=None*, *a=0*, *b=6.283185307179586*, *axes=None*, *ret_cubegrid=False*)

Arbitrary-dimension nice Fourier Transform.

This function wraps numpy's `fftn` and applies some nice properties. Notably, the returned fourier transform is equivalent to what would be expected from a continuous Fourier Transform (including normalisations etc.). In addition, arbitrary conventions are supported (see `powerbox.dft` for details).

Default parameters have the same normalising conventions as `numpy.fft.fftn`.

The output object always has the zero in the centre, with monotonically increasing spectral arguments.

Parameters

X [array] An array with arbitrary dimensions defining the field to be transformed. Should correspond exactly to the continuous function for which it is an analogue. A lower-dimensional transform can be specified by using the `axes` argument.

L [float or array-like, optional] The length of the box which defines `X`. If a scalar, each transformed dimension in `X` is assumed to have the same length. If array-like, must be of the same length as the number of transformed dimensions. The default returns the un-normalised DFT (same as numpy).

Lk [float or array-like, optional] The length of the fourier-space box which defines the dual of `X`. Only one of L/Lk needs to be provided. If provided, L takes precedence. If a scalar, each transformed dimension in `X` is assumed to have the same length. If array-like, must be of the same length as the number of transformed dimensions.

a,b [float, optional] These define the Fourier convention used. See `powerbox.dft` for details. The defaults return the standard DFT as defined in `numpy.fft`.

axes [sequence of ints, optional] The axes to take the transform over. The default is to use all axes for the transform.

ret_cubegrid [bool, optional] Whether to return the entire grid of frequency magnitudes.

Returns

ft [array] The DFT of X, normalised to be consistent with the continuous transform.

freq [list of arrays] The frequencies in each dimension, consistent with the Fourier conventions specified.

grid [array] Only returned if `ret_cubegrid` is `True`. An array with shape given by `axes` specifying the magnitude of the frequencies at each point of the fourier transform.

ifft

`powerbox.dft.ifft`(*X*, *Lk=None*, *L=None*, *a=0*, *b=6.283185307179586*, *axes=None*, *ret_cubegrid=False*)

Arbitrary-dimension nice inverse Fourier Transform.

This function wraps numpy's `ifftn` and applies some nice properties. Notably, the returned fourier transform is equivalent to what would be expected from a continuous inverse Fourier Transform (including normalisations etc.). In addition, arbitrary conventions are supported (see `powerbox.dft` for details).

Default parameters have the same normalising conventions as `numpy.fft.ifftn`.

Parameters

X [array] An array with arbitrary dimensions defining the field to be transformed. Should correspond exactly to the continuous function for which it is an analogue. A lower-dimensional transform can be specified by using the `axes` argument. Note that this should have its zero in the center.

Lk [float or array-like, optional] The length of the box which defines `X`. If a scalar, each transformed dimension in `X` is assumed to have the same length. If array-like, must be of the same length as the number of transformed dimensions. The default returns the un-normalised DFT (the same as numpy).

L [float or array-like, optional] The length of the real-space box, defining the dual of `X`. Only one of Lk/L needs to be passed. If L is passed, it is used. If a scalar, each transformed dimension in `X` is assumed to have the same length. If array-like, must be of the same length as the number of transformed dimensions. The default of `Lk=1` returns the un-normalised DFT.

a,b [float, optional] These define the Fourier convention used. See *`powerbox.dft`* for details. The defaults return the standard DFT as defined in `numpy.fft`.

axes [sequence of ints, optional] The axes to take the transform over. The default is to use all axes for the transform.

ret_cubegrid [bool, optional] Whether to return the entire grid of real-space co-ordinate magnitudes.

Returns

ft [array] The IDFT of X, normalised to be consistent with the continuous transform.

freq [list of arrays] The real-space co-ordinate grid in each dimension, consistent with the Fourier conventions specified.

grid [array] Only returned if `ret_cubegrid` is `True`. An array with shape given by `axes` specifying the magnitude of the real-space co-ordinates at each point of the inverse fourier transform.

fftfreq

`powerbox.dft.`**`fftfreq`**(*N*, *d=1.0*, *b=6.283185307179586*)

Return the fourier frequencies for a box with N cells, using general Fourier convention.

Parameters

N [int] The number of grid cells

d [float, optional] The interval between cells

b [float, optional] The fourier-convention of the frequency component (see *`powerbox.dft`* for details).

Returns

freq [array] The N symmetric frequency components of the Fourier transform. Always centred at 0.

fftshift

`powerbox.dft.`**`fftshift`**(*x*, **args*, ***kwargs*)

The same as numpy's fftshift, except that it preserves units (if Astropy quantities are used)

All extra arguments are passed directly to numpy's *fftshift*.

ifftshift

`powerbox.dft.`**`ifftshift`**(*x*, **args*, ***kwargs*)

The same as numpy's ifftshift, except that it preserves units (if Astropy quantities are used)

All extra arguments are passed directly to numpy's *ifftshift*.

5.6.3 powerbox.tools Module

A set of tools for dealing with structured boxes, such as those output by `powerbox`. Tools include those for averaging a field angularly, and generating the isotropic power spectrum.

Functions

`angular_average`(field, coords, bins[, ...])	Average a given field within radial bins.
`angular_average_nd`(field, coords, bins[, n, ...])	Average the first n dimensions of a given field within radial bins.
`get_power`(deltax, boxlength[, deltax2, N, ...])	Calculate the isotropic power spectrum of a given field, or cross-power of two similar fields.

angular_average

`powerbox.tools.`**`angular_average`**(*field*, *coords*, *bins*, *weights=1*, *average=True*, *bin_ave=True*, *get_variance=False*, *log_bins=False*)

Average a given field within radial bins.

This function can be used in fields of arbitrary dimension (memory permitting), and the field need not be centred at the origin. The averaging assumes that the grid cells fall completely into the bin which encompasses the co-ordinate point for the cell (i.e. there is no weighted splitting of cells if they intersect a bin edge).

It is optimized for applying a set of weights, and obtaining the variance of the mean, at the same time as averaging.

Parameters

field: nd-array An array of arbitrary dimension specifying the field to be angularly averaged.

coords: nd-array or list of n arrays. Either the *magnitude* of the co-ordinates at each point of *field*, or a list of 1D arrays specifying the co-ordinates in each dimension.

bins: float or array. The `bins` argument provided to histogram. Can be an int or array specifying radial bin edges.

weights: array, optional An array of the same shape as *field*, giving a weight for each entry.

average: bool, optional Whether to take the (weighted) average. If False, returns the (unweighted) sum.

bin_ave [bool, optional] Whether to return the bin co-ordinates as the (weighted) average of cells within the bin (if True), or the regularly spaced edges of the bins.

get_variance [bool, optional] Whether to also return an estimate of the variance of the power in each bin.

log_bins [bool, optional] Whether to create bins in log-space.

Returns

field_1d [1D-array] The angularly-averaged field.

bins [1D-array] Array of same shape as field_1d specifying the radial co-ordinates of the bins. Either the mean co-ordinate from the input data, or the regularly spaced bins, dependent on *bin_ave*.

var [1D-array, optional] The variance of the averaged field (same shape as bins), estimated from the mean standard error. Only returned if *get_variance* is True.

See also:

`angular_average_nd` Perform an angular average in a subset of the total dimensions.

Notes

If desired, the variance is calculated as the weight unbiased variance, using the formula at https://en.wikipedia.org/wiki/Weighted_arithmetic_mean#Reliability_weights for the variance in each cell, and normalising by a factor of V_2/V_1^2 to estimate the variance of the average.

Examples

Create a 3D radial function, and average over radial bins:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-5,5,128)   # Setup a grid
>>> X,Y,Z = np.meshgrid(x,x,x)
>>> r = np.sqrt(X**2+Y**2+Z**2) # Get the radial co-ordinate of grid
>>> field = np.exp(-r**2)       # Generate a radial field
>>> avgfunc, bins = angular_average(field,r,bins=100)   # Call angular_average
>>> plt.plot(bins, np.exp(-bins**2), label="Input Function")   # Plot input␣
↪function versus ang. avg.
>>> plt.plot(bins, avgfunc, label="Averaged Function")
```

angular_average_nd

powerbox.tools.**angular_average_nd**(*field*, *coords*, *bins*, *n=None*, *weights=1*, *average=True*, *bin_ave=True*, *get_variance=False*, *log_bins=False*)

Average the first n dimensions of a given field within radial bins.

This function be used to take "hyper-cylindrical" averages of fields. For a 3D field, with *n=2*, this is exactly a cylindrical average. This function can be used in fields of arbitrary dimension (memory permitting), and the field need not be centred at the origin. The averaging assumes that the grid cells fall completely into the bin which encompasses the co-ordinate point for the cell (i.e. there is no weighted splitting of cells if they intersect a bin edge).

It is optimized for applying a set of weights, and obtaining the variance of the mean, at the same time as averaging.

Parameters

field [md-array] An array of arbitrary dimension specifying the field to be angularly averaged.

coords [list of n arrays] A list of 1D arrays specifying the co-ordinates in each dimension *to be average*.

bins [int or array.] Specifies the radial bins for the averaged dimensions. Can be an int or array specifying radial bin edges.

n [int, optional] The number of dimensions to be averaged. By default, all dimensions are averaged. Always uses the first *n* dimensions.

weights [array, optional] An array of the same shape as the first *n* dimensions of *field*, giving a weight for each entry.

average [bool, optional] Whether to take the (weighted) average. If False, returns the (unweighted) sum.

bin_ave [bool, optional] Whether to return the bin co-ordinates as the (weighted) average of cells within the bin (if True), or the linearly spaced edges of the bins

get_variance [bool, optional] Whether to also return an estimate of the variance of the power in each bin.

log_bins [bool, optional] Whether to create bins in log-space.

Returns

field [(m-n+1)-array] The angularly-averaged field. The first dimension corresponds to *bins*, while the rest correspond to the unaveraged dimensions.

bins [1D-array] The radial co-ordinates of the bins. Either the mean co-ordinate from the input data, or the regularly spaced bins, dependent on *bin_ave*.

var [(m-n+1)-array, optional] The variance of the averaged field (same shape as *field*), estimated from the mean standard error. Only returned if *get_variance* is True.

Examples

Create a 3D radial function, and average over radial bins. Equivalent to calling `angular_average()`:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-5,5,128)   # Setup a grid
>>> X,Y,Z = np.meshgrid(x,x,x)  # ""
>>> r = np.sqrt(X**2+Y**2+Z**2) # Get the radial co-ordinate of grid
>>> field = np.exp(-r**2)       # Generate a radial field
>>> avgfunc, bins, _ = angular_average_nd(field,[x,x,x],bins=100)   # Call␣
↪angular_average
>>> plt.plot(bins, np.exp(-bins**2), label="Input Function")   # Plot input␣
↪function versus ang. avg.
>>> plt.plot(bins, avgfunc, label="Averaged Function")
```

Create a 2D radial function, extended to 3D, and average over first 2 dimensions (cylindrical average):

```
>>> r = np.sqrt(X**2+Y**2)
>>> field = np.exp(-r**2)    # 2D field
>>> field = np.repeat(field,len(x)).reshape((len(x),)*3)   # Extended to 3D
>>> avgfunc, avbins, coords = angular_average_nd(field, [x,x,x], bins=50, n=2)
>>> plt.plot(avbins, np.exp(-avbins**2), label="Input Function")
>>> plt.plot(avbins, avgfunc[:,0], label="Averaged Function")
```

get_power

`powerbox.tools.`**`get_power`**`(`*deltax*, *boxlength*, *deltax2=None*, *N=None*, *a=1.0*, *b=1.0*, *remove_shotnoise=True*, *vol_normalised_power=True*, *bins=None*, *res_ndim=None*, *weights=None*, *weights2=None*, *dimensionless=True*, *bin_ave=True*, *get_variance=False*, *log_bins=False*, *ignore_zero_mode=False*, *k_weights=1*`)`

Calculate the isotropic power spectrum of a given field, or cross-power of two similar fields.

This function, by default, conforms to typical cosmological power spectrum conventions – normalising by the volume of the box and removing shot noise if applicable. These options are configurable.

Parameters

deltax [array-like] The field on which to calculate the power spectrum . Can either be arbitrarily n-dimensional, or 2-dimensional with the first being the number of spatial dimensions, and the second the positions of discrete particles in the field. The former should represent a density field, while the latter is a discrete sampling of a field. This function chooses which to use by checking the value of *N* (see below). Note that if a discrete sampling is used, the power spectrum calculated is the "overdensity" power spectrum, i.e. the field re-centered about zero and rescaled by the mean.

boxlength [float or list of floats] The length of the box side(s) in real-space.

deltax2 [array-like] If given, a box of the same shape as deltax, against which deltax will be cross correlated.

N [int, optional] The number of grid cells per side in the box. Only required if deltax is a discrete sample. If given, the function will assume a discrete sample.

a,b [float, optional] These define the Fourier convention used. See `powerbox.dft` for details. The defaults define the standard usage in *cosmology* (for example, as defined in Cosmological Physics, Peacock, 1999, pg. 496.). Standard numerical usage (eg. numpy) is (a,b) = (0,2pi).

remove_shotnoise [bool, optional] Whether to subtract a shot-noise term after determining the isotropic power. This only affects discrete samples.

vol_weighted_power [bool, optional] Whether the input power spectrum, `pk`, is volume-weighted. Default True because of standard cosmological usage.

bins [int or array, optional] Defines the final k-bins output. If None, chooses a number based on the input resolution of the box. Otherwise, if int, this defines the number of kbins, or if an array, it defines the exact bin edges.

res_ndim [int, optional] Only perform angular averaging over first *res_ndim* dimensions. By default, uses all dimensions.

weights, weights2 [array-like, optional] If deltax is a discrete sample, these are weights for each point.

dimensionless: bool, optional Whether to normalise the cube by its mean prior to taking the power.

bin_ave [bool, optional] Whether to return the bin co-ordinates as the (weighted) average of cells within the bin (if True), or the linearly spaced edges of the bins

get_variance [bool, optional] Whether to also return an estimate of the variance of the power in each bin.

log_bins [bool, optional] Whether to create bins in log-space.

ignore_zero_mode [bool, optional] Whether to ignore the k=0 mode (or DC term).

k_weights [nd-array, optional] The weights of the n-dimensional k modes. This can be used to filter out some modes completely.

Returns

p_k [array] The power spectrum averaged over bins of equal $|k|$.

meank [array] The bin-centres for the p_k array (in k). This is the mean k-value for cells in that bin.

var [array] The variance of the power spectrum, estimated from the mean standard error. Only returned if *get_variance* is True.

Examples

One can use this function to check whether a box created with `PowerBox` has the correct power spectrum:

```
>>> from powerbox import PowerBox
>>> import matplotlib.pyplot as plt
>>> pb = PowerBox(250,lambda k : k**-2.)
>>> p,k = get_power(pb.delta_x,pb.boxlength)
>>> plt.plot(k,p)
>>> plt.plot(k,k**-2.)
>>> plt.xscale('log')
>>> plt.yscale('log')
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index