powerbox Documentation

Release 0.5.3

Steven Murray

Jul 19, 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

Clone/Download then `python setup.py install`. Or just `pip install powerbox`.

CHAPTER 3

Acknowledgment

If you find `powerbox` useful in your research, please cite http://ascl.net/1805.001

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

```
In [3]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
```

There are two useful classes in powerbox: the basic PowerBox, and one for log-normal fields: LogNormalPowerBox. You can import them like this:

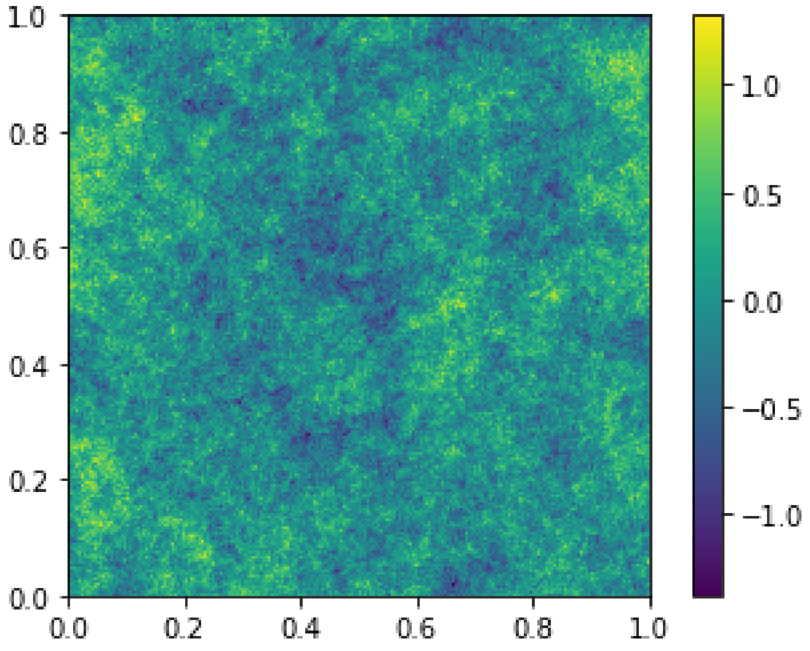
```
In [1]: from powerbox import PowerBox, LogNormalPowerBox
```

Once imported, to see all the options, just use help(PowerBox).

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

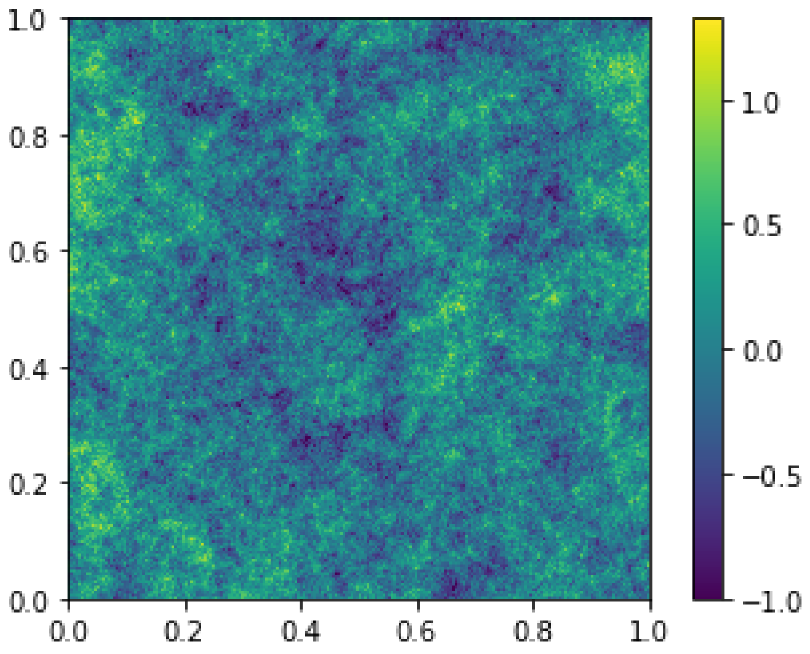
For a basic 2D Gaussian field with a power-law power-spectrum, one can use the following:

```
In [10]: pb = 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

         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 [11]: pb = 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
                       ensure_physical=True)           # ** Ensure the delta_x is a physically valid over-

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

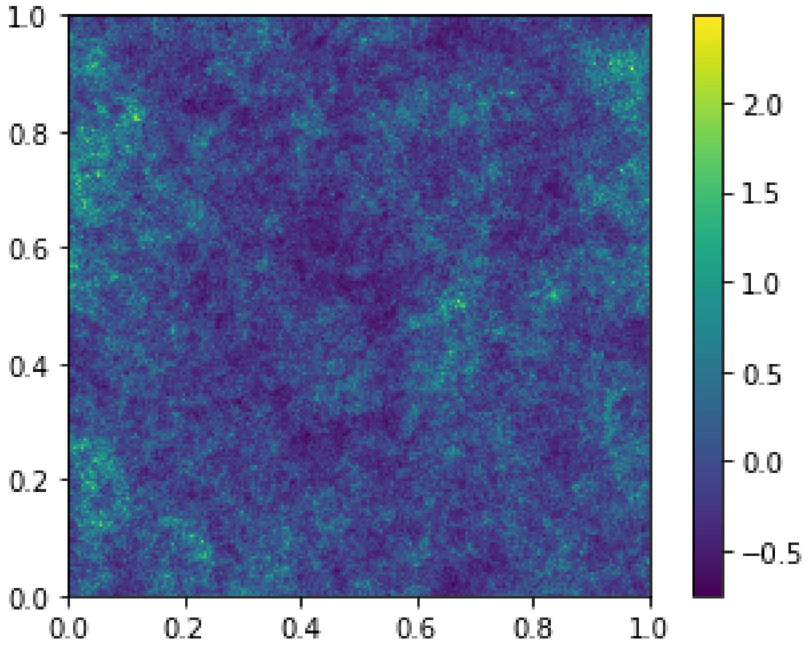


If you are actually dealing with over-densities, then this clipping solution is probably a bit hacky. What you want is a log-normal field…

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 [12]: lnpb = 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
                                  seed = 1010)                 # Use the same seed as our powerbox

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

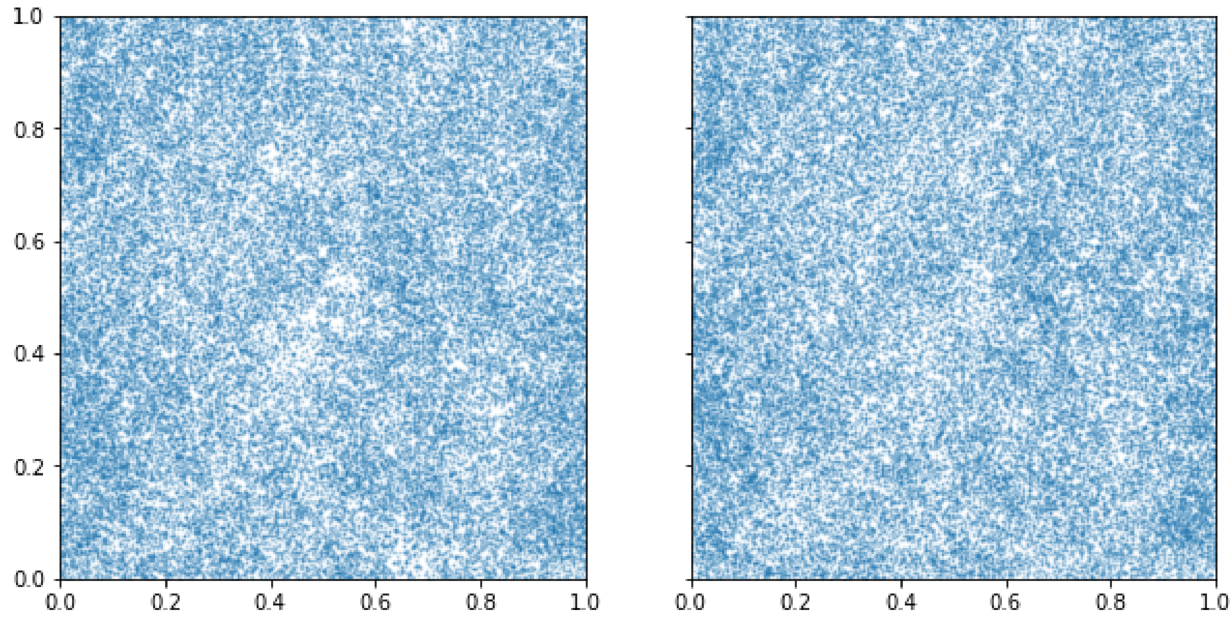


Again, the `delta_x` is zero-mean, but has a longer positive tail due to the log-normal nature of the distribution. This means it is always greater than -1, so that the over-density field is always physical.

Create some discrete samples on the field

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

```
In [18]: fig, ax = plt.subplots(1,2, sharex=True,sharey=True,gridspec_kw={"hspace":0}, subplot_kw={"y

         # 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

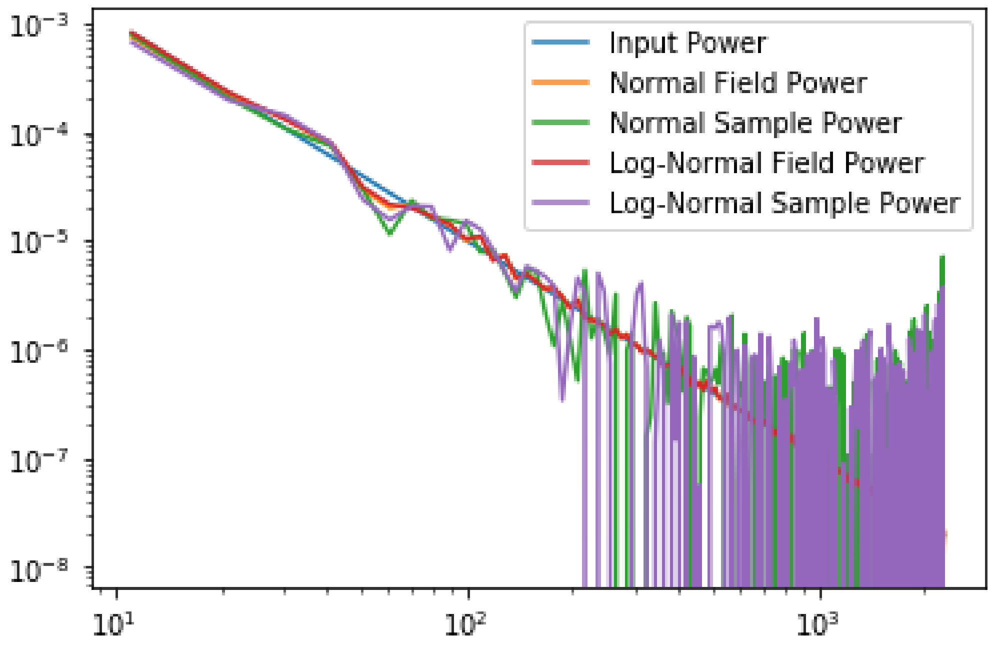
`powerbox` also contains a function for computing the (isotropic) power-spectrum of a field. This function accepts either a box defining the field values at every co-ordinate, *or* a set of discrete samples. In the latter case, the routine returns the power spectrum of over-densities, which matches the 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 [19]: from powerbox import get_power
```

```
In [24]: # 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 [26]: 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 Create a log-normal mock dark-matter distribution

```
In [3]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
```

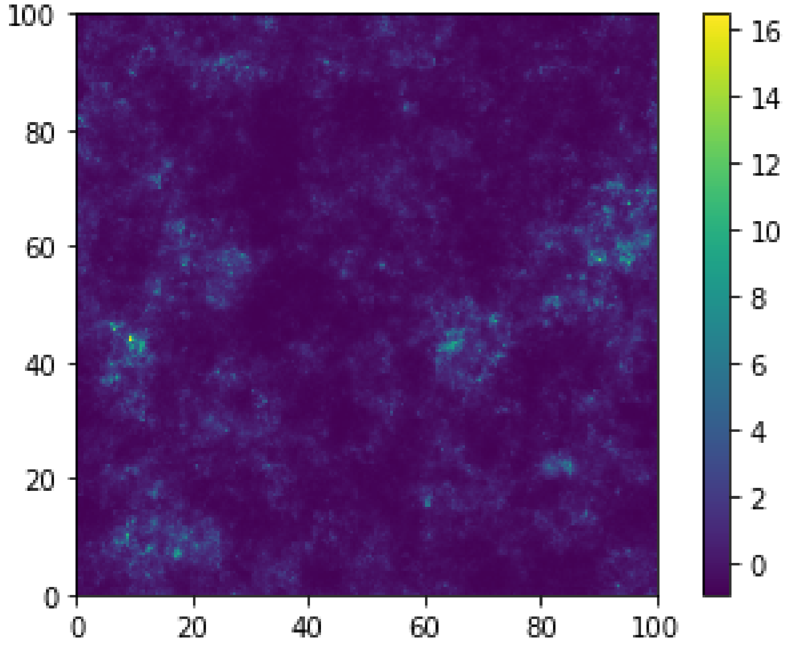
In this demo, we create a mock dark-matter distribution, based on the cosmological power spectrum. To generate the power-spectrum we use the `hmf` code (https://github.com/steven-murray/hmf).

The box can be set up like this:

```
In [5]: from hmf import MassFunction
        from scipy.interpolate import InterpolatedUnivariateSpline as spline
        import numpy as np
        from powerbox import LogNormalPowerBox

        # Set up a MassFunction instance to access its cosmological power-spectrum
        mf = 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 = LogNormalPowerBox(N=256, dim=3, pk = power, boxlength= 100.)
```

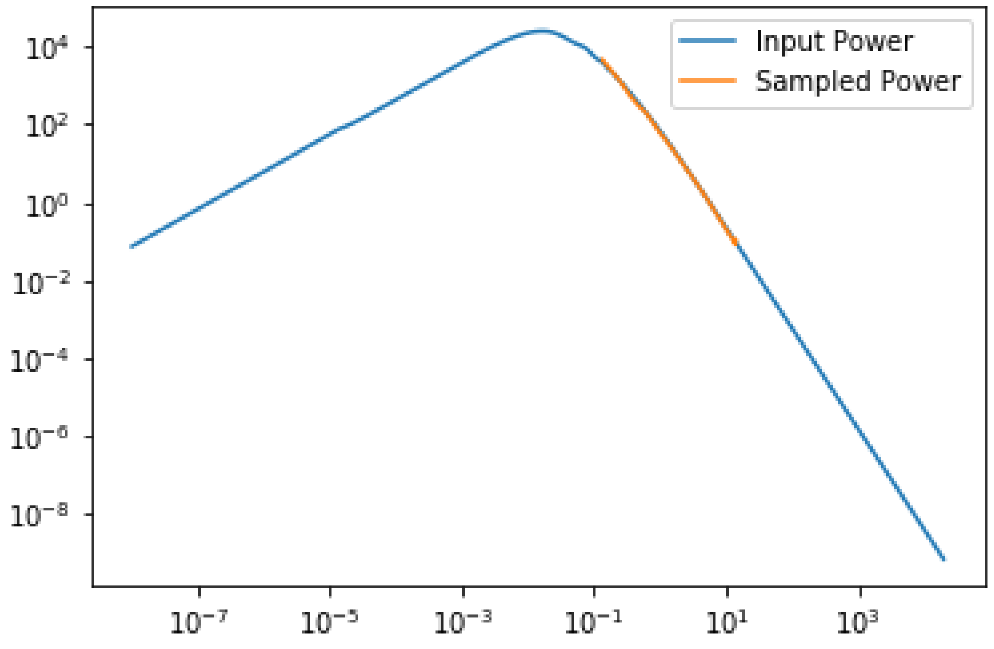
Now we can make a plot of a slice of the density field:

```
In [6]: plt.imshow(np.mean(pb.delta_x[:100,:,:],axis=0),extent=(0,100,0,100))
        plt.colorbar()
        plt.show()
```



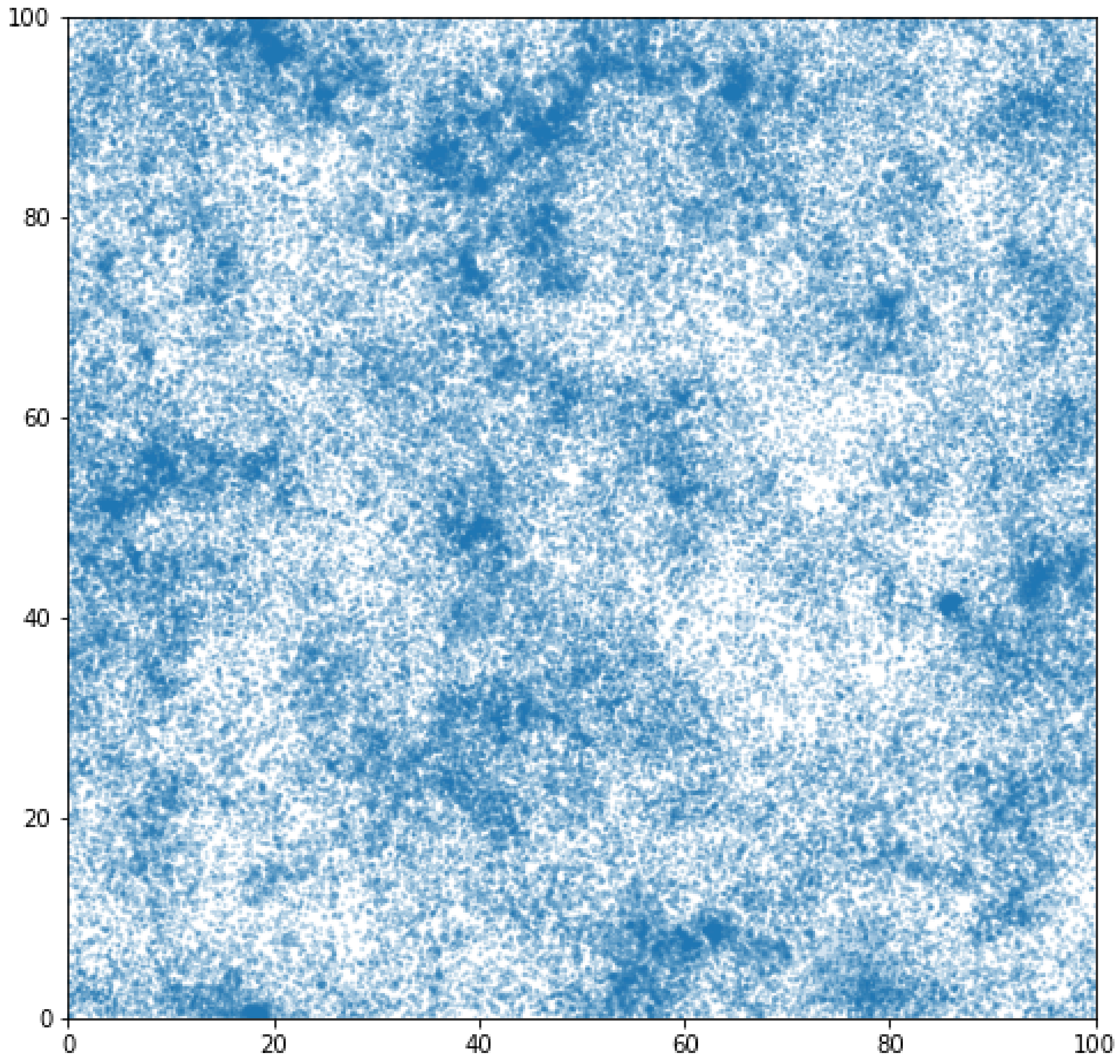
And we can also compare the power-spectrum of the output field to the input power:

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

        p_k, kbins = get_power(pb.delta_x,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()
```



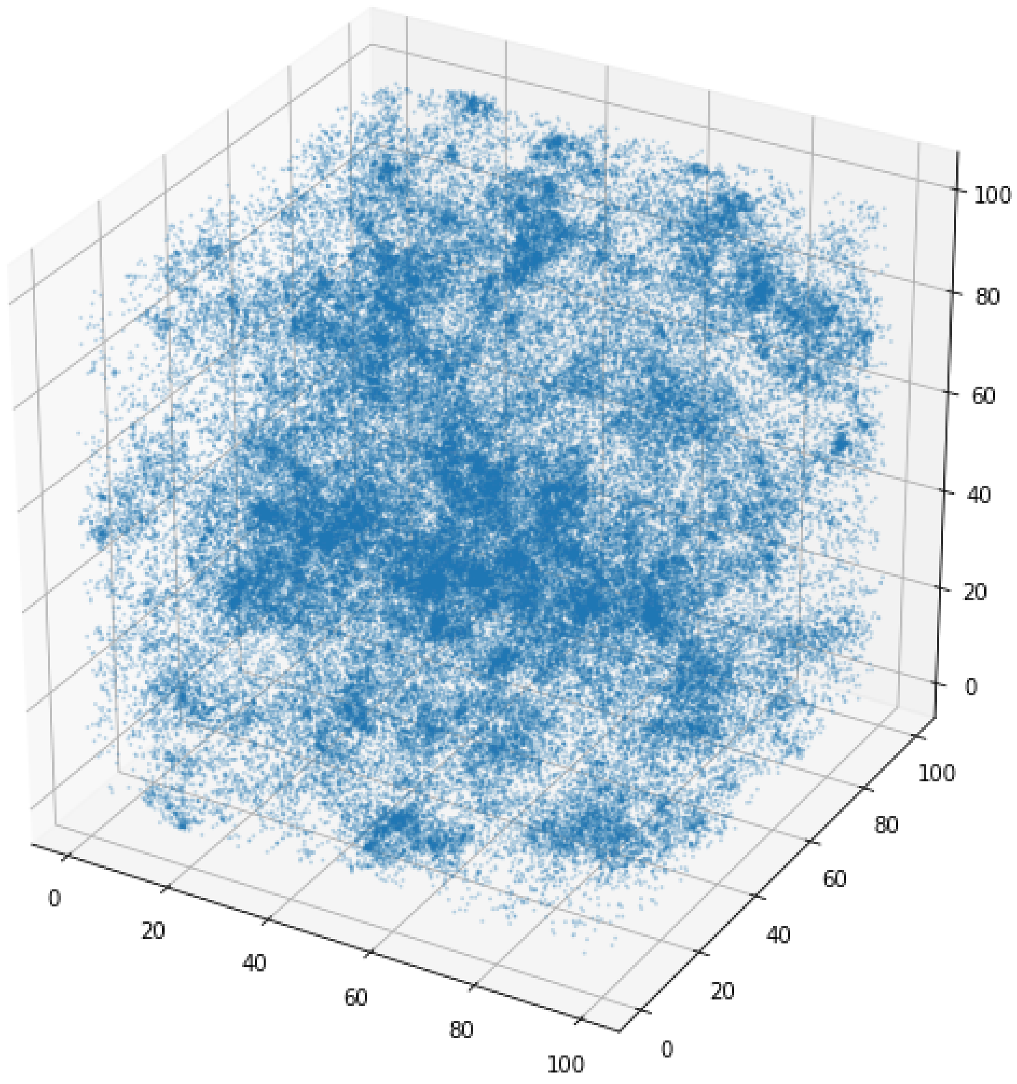
Furthermore, we can sample a set of discrete particles on the field and plot them:

```
In [10]: particles = pb.create_discrete_sample(nbar=0.1,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(0,100)
         plt.ylim(0,100)
         plt.show()
```



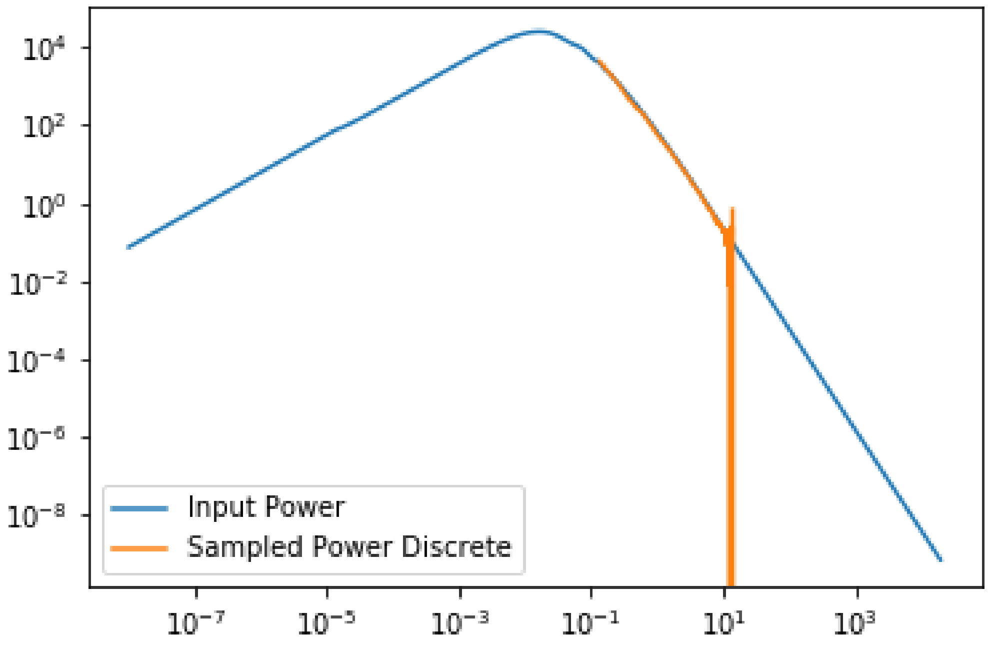
Or plot them in 3D!

```
In [17]: 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 [19]: p_k_sample, kbins_sample = 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()
```



5.1.3 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:

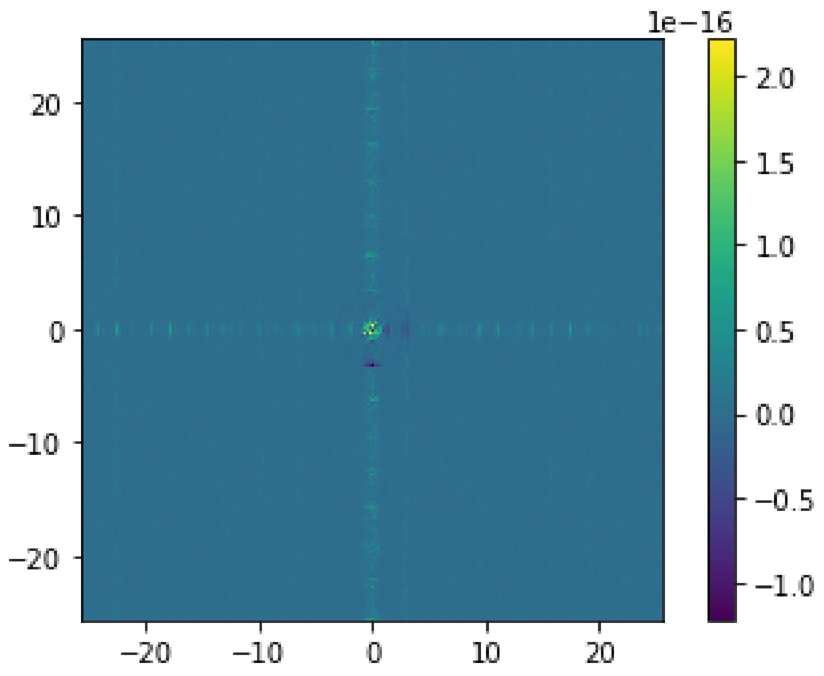
$$\int e^{-\pi r^2} e^{-2\pi i k \cdot x} d^2x = e^{-\pi k^2},$$

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

```
In [2]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np

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

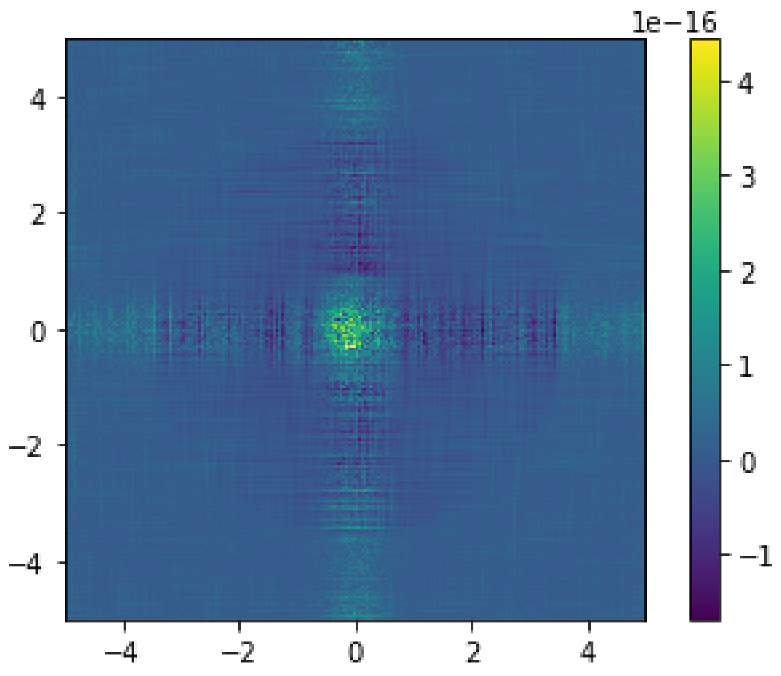
```
In [7]: # Parameters of the field
        L = 10.
        N = 512
        dx = L/N

        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()
```

```
Out[7]: <matplotlib.colorbar.Colorbar at 0x7fcedc018690>
```



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

```
In [10]: 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.colorbar()
         plt.show()
```



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

```
In [11]: x_ -x
```

```
Out[11]: array([[ 0.,   0.,   0., ...,   0.,   0.,   0.],
               [ 0.,   0.,   0., ...,   0.,   0.,   0.]])
```

Changing the convention

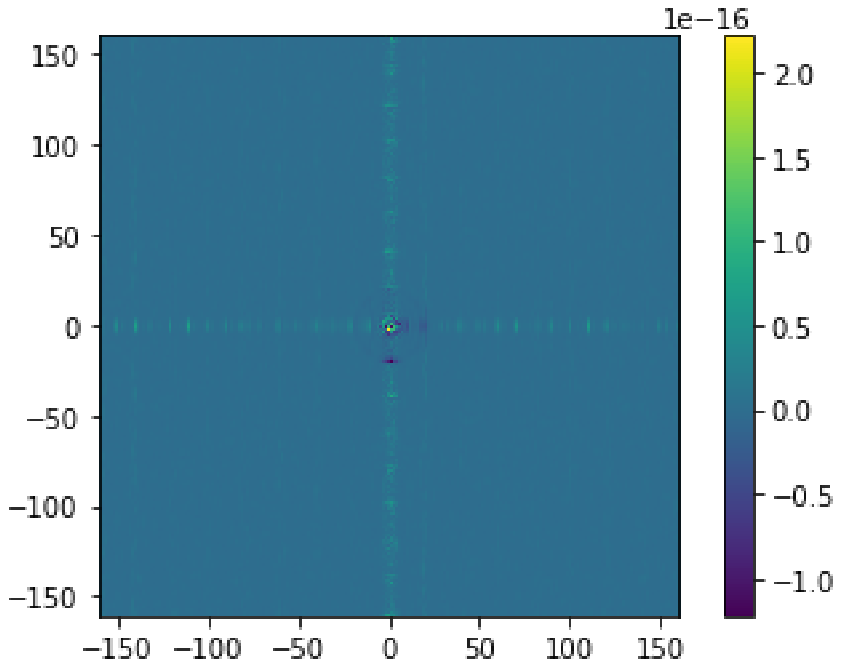
Suppose we instead required the transform

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

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

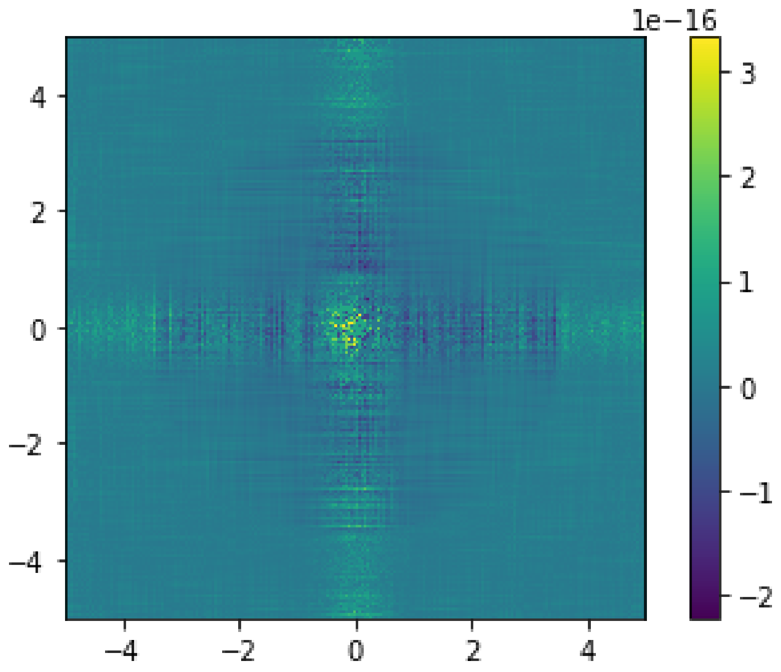
```
In [14]: # Generate the k-space field, the 1D k-space grid, and the 2D magnitude grid.
         k_field, k, rk = fft(field,L=L,           # Pass the field to transform, and its size
                              ret_cubegrid=True,    # Tell it to return the grid of magnitudes.
                              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()
```

```
Out[14]: <matplotlib.colorbar.Colorbar at 0x7fcec6148e50>
```



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

```
In [15]: 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

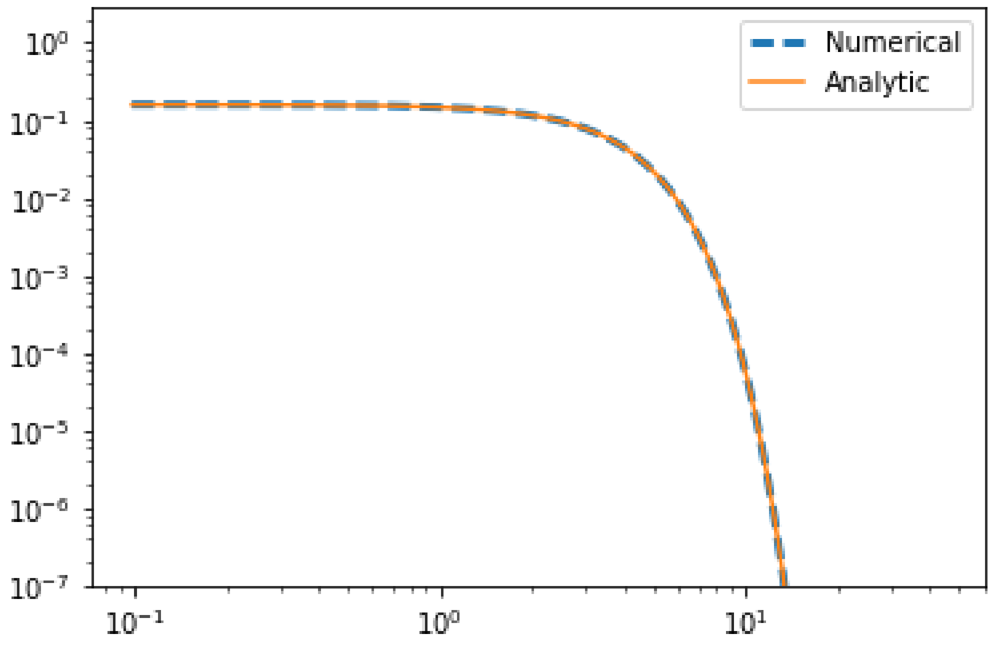
$$f(r/(2\pi))/2\pi,$$

and the inverse-then-forward should be

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

```
In [21]: # Import a handy function to do radial binning
         from powerbox import angular_average

         # 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 = 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()
```



Using Different Conventions in Powerbox

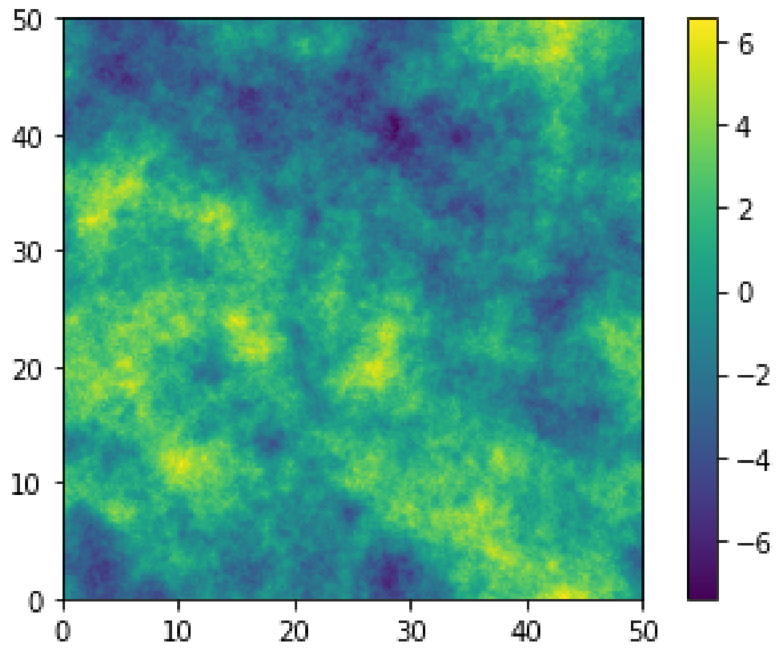
These fourier-transform wrappers are used inside powerbox to do the heavy lifting. That means that one can pass a power spectrum which has been defined with arbitrary conventions, and receive a fully consistent box back.

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

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

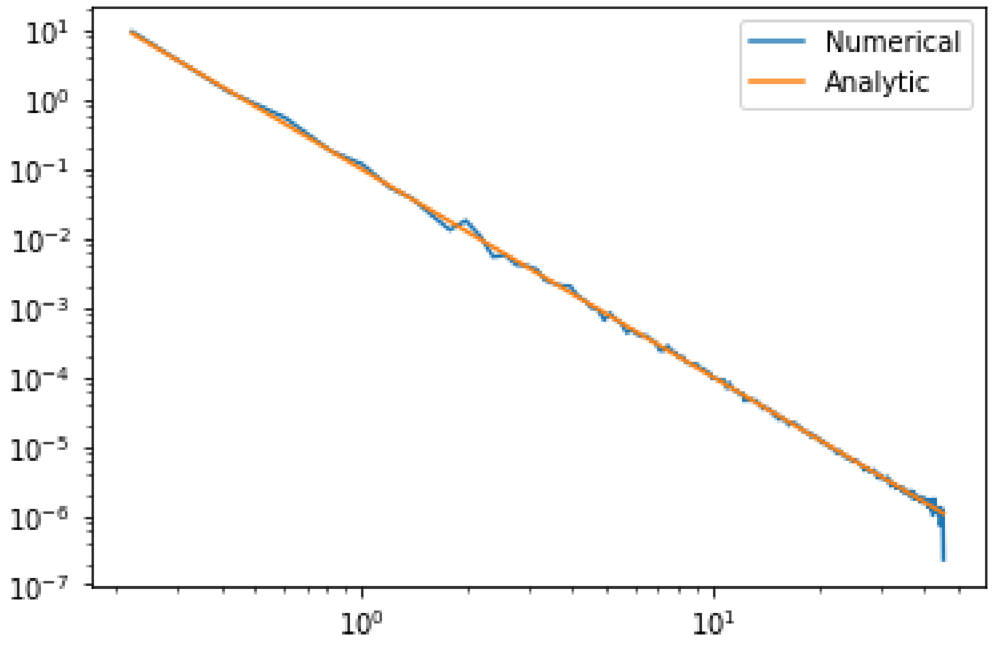
We now wish to create a realisation with a power spectrum following these conventions. Let's say the power spectrum is $P(k) = 0.1k^{-2}$.

```
In [39]: from powerbox import PowerBox

         pb = 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 [40]: from powerbox import get_power

         power, kbins = 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 v0.5.3 [22 May 2018]

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

5.3.2 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.

5.3.3 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.

v0.4.3 [29 March 2017]

Bugfixes - Fixed volume normalisation in get_power.

v0.4.2 [28 March 2017]

Features - Added ability to cross-correlate boxes in get_power.

v0.4.1

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

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

v0.3.2

Bugfixes - Fixed bug in pyFFTW cache setting

v0.3.1

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

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

v0.2.3 [11 Jan 2017]

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

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

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.

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.

v0.1.0 [27 Oct 2016]

First working version. Only Gaussian fields working.

5.4 API Summary

powerbox.powerbox	The main module of `powerbox`.
powerbox.tools	A set of tools for dealing with structured boxes, such as those output by `powerbox`.
powerbox.dft	A module defining some “nicer” fourier transform functions.

5.4.1 powerbox.powerbox

The main module of `powerbox`. Provides classes to create structured boxes.

Classes

LogNormalPowerBox(*args, **kwargs)	A subclass of *PowerBox* which calculates Log-Normal density fields with given power spectra.
PowerBox(N, pk[, dim, boxlength, . . .])	An object which calculates and stores the real-space and fourier-space fields generated with a given power spectrum.

powerbox.powerbox.LogNormalPowerBox

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

A subclass of *PowerBox* which calculates 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)
```

Methods

`__init__`(*args, **kwargs)	x.__init__(...) initializes x; see help(type(x)) for signature
`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*

powerbox.powerbox.LogNormalPowerBox.__init__

`LogNormalPowerBox.`**`__init__`**(**args*, ***kwargs*)

x.__init__(...) initializes x; see help(type(x)) for signature

powerbox.powerbox.LogNormalPowerBox.correlation_array

`LogNormalPowerBox.`**`correlation_array`**()

The correlation function from the input power, on the grid

powerbox.powerbox.LogNormalPowerBox.create_discrete_sample

`LogNormalPowerBox.`**`create_discrete_sample`**(*nbar*, *randomise_in_cell=True*, *min_at_zero=False*, *store_pos=False*, *seed=None*)

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.

powerbox.powerbox.LogNormalPowerBox.delta_k

`LogNormalPowerBox.`**`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)

powerbox.powerbox.LogNormalPowerBox.delta_x

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

powerbox.powerbox.LogNormalPowerBox.gauss_hermitian

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

powerbox.powerbox.LogNormalPowerBox.gaussian_correlation_array

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

powerbox.powerbox.LogNormalPowerBox.gaussian_power_array

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

powerbox.powerbox.LogNormalPowerBox.k

`LogNormalPowerBox.`**`k`**`()`
The entire grid of wavenumber magitudes

powerbox.powerbox.LogNormalPowerBox.power_array

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

Attributes

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

powerbox.powerbox.LogNormalPowerBox.kvec

`LogNormalPowerBox.`**`kvec`**
The vector of wavenumbers along a side

powerbox.powerbox.LogNormalPowerBox.r

`LogNormalPowerBox.`**`r`**
The radial position of every point in the grid

powerbox.powerbox.LogNormalPowerBox.x

`LogNormalPowerBox.`**`x`**
The co-ordinates of the grid along a side

powerbox.powerbox.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*)
An object which calculates and stores the real-space and fourier-space 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 [func] A function 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 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 a log-normal box.

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

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, in line with Mathematica's definition.

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.

Examples

To create a 3-dimensional box of gaussian over-densities, with side length 1 Mpc, gridded equally into 100 bins, and where k=2pi/x, with 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)
```

Methods

`__init__`(N, pk[, dim, boxlength, ...])	x.__init__(...) initializes x; see help(type(x)) for signature
`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*

powerbox.powerbox.PowerBox.__init__

PowerBox.**__init__**(*N*, *pk*, *dim=2*, *boxlength=1.0*, *ensure_physical=False*, *a=1.0*, *b=1.0*, *vol_normalised_power=True*, *seed=None*)

x.__init__(...) initializes x; see help(type(x)) for signature

powerbox.powerbox.PowerBox.create_discrete_sample

PowerBox.**create_discrete_sample**(*nbar*, *randomise_in_cell=True*, *min_at_zero=False*, *store_pos=False*, *seed=None*)

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.

powerbox.powerbox.PowerBox.delta_k

`PowerBox.`**`delta_k`**`()`

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

powerbox.powerbox.PowerBox.delta_x

`PowerBox.`**`delta_x`**`()`

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

powerbox.powerbox.PowerBox.gauss_hermitian

`PowerBox.`**`gauss_hermitian`**`()`

A random array which has Gaussian magnitudes and Hermitian symmetry

powerbox.powerbox.PowerBox.k

`PowerBox.`**`k`**`()`

The entire grid of wavenumber magitudes

powerbox.powerbox.PowerBox.power_array

`PowerBox.`**`power_array`**`()`

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

Attributes

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

powerbox.powerbox.PowerBox.kvec

`PowerBox.`**`kvec`**

The vector of wavenumbers along a side

powerbox.powerbox.PowerBox.r

`PowerBox.`**`r`**

The radial position of every point in the grid

powerbox.powerbox.PowerBox.x

`PowerBox.`**`x`**

The co-ordinates of the grid along a side

5.4.2 powerbox.tools

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

Functions

`angular_average`(field, coords, bins[, …])	Perform a radial histogram – averaging within radial bins – of a field.
`angular_average_nd`(field, coords, bins[, n, …])	Take an ND box, and perform a radial average over the first n dimensions.
`get_power`(deltax, boxlength[, deltax2, N, …])	Calculate the isotropic power spectrum of a given field.

powerbox.tools.angular_average

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

Perform a radial histogram – averaging within radial bins – of a field.

Parameters

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

coords [array] The magnitude of the co-ordinates at each point of *field*. Must be the same size as field.

bins [float or array.] The `bins` argument provided to histogram. Can be an int or array specifying 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 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.

Returns

field_1d [array] The field averaged angularly (finally 1D)

binavg [array] The mean co-ordinate in each radial bin.

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

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 estimated 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")
```

powerbox.tools.angular_average_nd

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

Take an ND box, and perform a radial average over the first n dimensions.

Parameters

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

coords [list of arrays] A list with length equal to the number of dimensions of *field*. Each entry should be an array specifying the co-ordinates in the corresponding dimension of *field*. Note this is different from `angular_average()`.

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

Returns

field_1d [array] The field averaged angularly (finally 1D)

bins [array] The mean co-ordinate in each radial bin (or the bin edges, if *bin_ave* is False)

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:

```
>>> 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")
```

powerbox.tools.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*)

Calculate the isotropic power spectrum of a given field.

Parameters

deltax [array-like] The field to calculate the power spectrum of. 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.

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')
```

5.4.3 powerbox.dft

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.

Notes

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.
`fftfreq`(N[, d, b])	Return the fourier frequencies for a box with N cells, using general Fourier convention.
`fftshift`(x, *args, **kwargs)	
`ifft`(X[, Lk, L, a, b, axes, ret_cubegrid])	Arbitrary-dimension nice inverse Fourier Transform.
`ifftshift`(x, *args, **kwargs)	

powerbox.dft.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 return exactly what `numpy.fft.fftn` would return.

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.

powerbox.dft.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.

powerbox.dft.fftshift

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

powerbox.dft.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 return exactly what `numpy.fft.ifftn` would return.

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.

powerbox.dft.ifftshift

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

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index