xrft Documentation

Release 0.1

xrft developers

Jun 13, 2022

GETTING STARTED

xrft is a Python package for taking the discrete Fourier transform (DFT) on xarray and dask arrays. It is:

- **Powerful**: It keeps the metadata and coordinates of the original xarray dataset and provides a clean work flow of DFT.
- **Easy-to-use**: It uses the native arguments of numpy FFT and provides a simple, high-level API.
- **Fast**: It uses the dask API of FFT and map_blocks to allow parallelization of DFT.

Note: xrft is at early stage of development and will keep improving in the future. The discrete Fourier transform API should be quite stable, but minor utilities could change in the next version. If you find any bugs or would like to request any enhancements, please raise an issue on GitHub.

CHAPTER

ONE

DOCUMENTATION

1.1 Overview: Why xrft?

1.1.1 For robustness and efficiency

In the field of Earth Science, we often take Fourier transforms of the variable of interest. There has, however, not been an universal algorithm in which we calculate the transforms and our aim is to stream line this process.

We utilize the dask API to parallelize the computation to make it efficient for large data sets.

1.1.2 For usability and simplicity

The arguments in xrft rely on well-estabilished standards (dask and numpy), so users don't need to learn a bunch of new syntaxes or even a new software stack.

xrft can track the metadata in `xarray.DataArray` (*example*), which makes it easy for large data sets.

The choice of Python and Anaconda also makes xrft *extremely easy to install*.

1.2 Current limitations

1.2.1 Discrete sinusoid transform

xrft currently only supports discrete fourier transforms. We plan to implement discrete sinusoid tranforms in the near future.

1.3 Installation

1.3.1 The quickest way

xrft is compatible both with Python 2 and 3. The major dependencies are xarray and dask. The best way to install them is using Anaconda:

```
$ conda install -c conda-forge xarray dask xrft .
```

It is also possible to install from PyPI by:

```
$ pip install xrft .
```

1.3.2 Install xrft from GitHub repo

To get the latest version:

```
$ git clone https://github.com/xgcm/xrft.git
$ cd xrft
$ python setup.py install .
```

Developers can track source code changes by:

```
$ git clone https://github.com/xgcm/xrft.git
$ cd xrft
$ python setup.py develop .
```

1.4 Contributor Guide

xrft is meant to be a community driven package and we welcome feedback and contributions.

Did you notice a bug? Are you missing a feature? A good first starting place is to open an issue in the github issues page.

In order to contribute to xrft, please fork the repository and submit a pull request. A good step by step tutorial for this can be found in the xarray contributor guide.

1.4.1 Environments

The easiest way to start developing xrft pull requests, is to install one of the conda environments provided in the ci folder:

```
conda env create -f ci/environment-py3.8.yml
```

Activate the environment with:

```
conda activate test_env_xrft
```

1.4.2 Code Formatting

We use black as code formatter and pull request will fail in the CI if not properly formatted.

All conda environments contain black and you can reformat code using:

```
black xrft
```

pre-commit provides an automated way to reformat your code prior to each commit. Simply install pre-commit:

```
pip install pre-commit
```

and install it in the xrft root directory with:

```
pre-commit install
```

and your code will be properly formatted before each commit.

1.4.3 How to release a new version of xrft (for maintainers only)

The process of releasing at this point is very easy.

We need only two things: A PR to update the documentation and and making a release on github.

1. Make sure that all the new features/bugfixes etc are appropriately documented in *doc/whats-new.rst*, add the date to the current release and make an empty (unreleased) entry for the next minor release as a PR.
2. Navigate to the 'tags' symbol on the repos main page, click on 'Releases' and on 'Draft new release' on the right. Add the version number and a short description and save the release.

From here the github actions take over and package things for Pypi. The conda-forge package will be triggered by the Pypi release and you will have to approve a PR in xrft-feedstock. This takes a while, usually a few hours to a day.

Thats it!

1.5 Example of discrete and inverse discrete Fourier transform

[1]:
```
import numpy as np
import numpy.testing as npt
import xarray as xr
import xrft
import numpy.fft as npft
import scipy.signal as signal
import dask.array as dsar
import matplotlib.pyplot as plt
%matplotlib inline
```

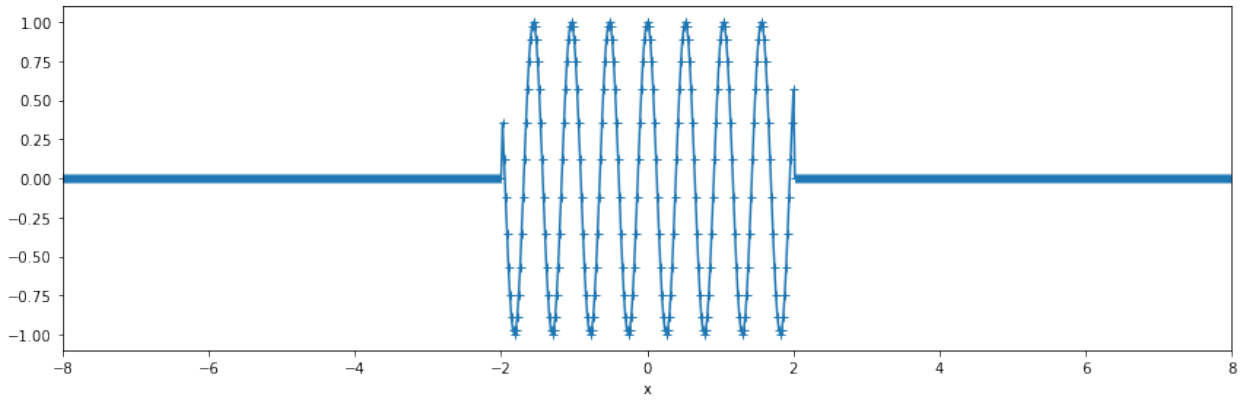
In this notebook, we provide examples of the discrete Fourier transform (DFT) and its inverse, and how `xrft` automatically harnesses the metadata. We compare the results to conventional `numpy.fft` (hereon `npft`) to highlight the strengths of `xrft`.

1.5.1 A case with synthetic data

Generate synthetic data centered around zero

[2]:
```
k0 = 1/0.52
T = 4.
dx = 0.02
x = np.arange(-2*T,2*T,dx)
y = np.cos(2*np.pi*k0*x)
y[np.abs(x)>T/2]=0.
da = xr.DataArray(y, dims=('x',), coords={'x':x})
```

[22]:
```
fig, ax = plt.subplots(figsize=(12,4))
fig.set_tight_layout(True)
da.plot(ax=ax, marker='+', label='original signal')
ax.set_xlim([-8,8]);
```



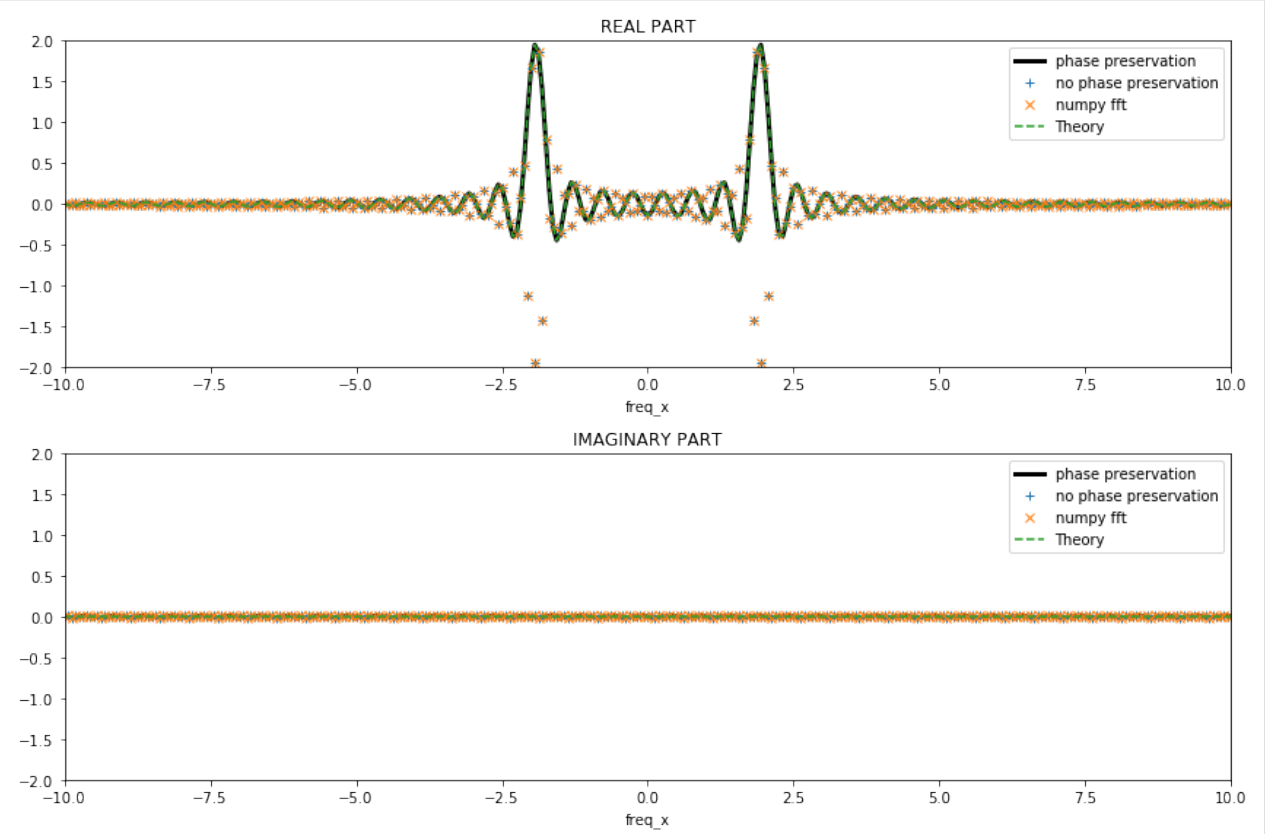
Let's take the Fourier transform

We will compare the Fourier transform with and without taking into consideration about the phase information.

[3]:
```
da_dft = xrft.dft(da, true_phase=True, true_amplitude=True) # Fourier Transform w/␣
↪consideration of phase
da_fft = xrft.fft(da)                                       # Fourier Transform w/ numpy.
↪fft-like behavior
da_npft = npft.fft(da)
```

[4]:
```
k = da_dft.freq_x # wavenumber axis
TF_s = T/2*(np.sinc(T*(k-k0)) + np.sinc(T*(k+k0))) # Theoretical result of the Fourier␣
↪transform
```

[26]:
```
fig, (ax1,ax2) = plt.subplots(figsize=(12,8), nrows=2, ncols=1)
fig.set_tight_layout(True)

(da_dft.real).plot(ax=ax1, linestyle='-', lw=3, c='k', label='phase preservation')
((da_fft*dx).real).plot(ax=ax1, linestyle='', marker='+',label='no phase preservation')
ax1.plot(k, (npft.fftshift(da_npft)*dx).real, linestyle='', marker='x',label='numpy fft')
ax1.plot(k, TF_s.real, linestyle='--', label='Theory')
ax1.set_xlim([-10,10])
ax1.set_ylim([-2,2])
ax1.legend()
ax1.set_title('REAL PART')

(da_dft.imag).plot(ax=ax2, linestyle='-', lw=3, c='k', label='phase preservation')
((da_fft*dx).imag).plot(ax=ax2, linestyle='', marker='+', label='no phase preservation')
ax2.plot(k, (npft.fftshift(da_npft)*dx).imag, linestyle='', marker='x',label='numpy fft')
ax2.plot(k, TF_s.imag, linestyle='--', label='Theory')
ax2.set_xlim([-10,10])
ax2.set_ylim([-2,2])
ax2.legend()
ax2.set_title('IMAGINARY PART');
```



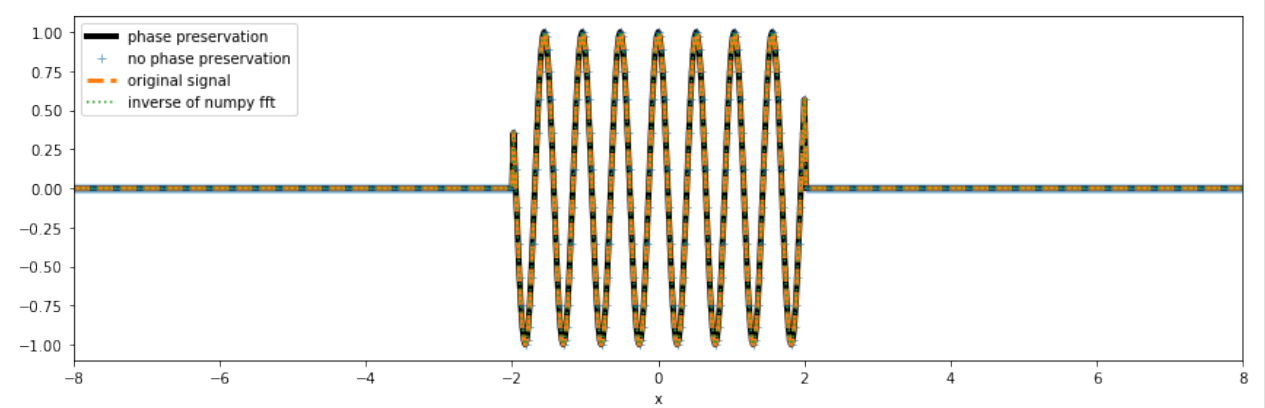
`xrft.dft`, `xrft.fft` (and `npft.fft` with careful `npft.fftshift`ing) all give the same amplitudes as theory (as the coordinates of the original data was centered) but the latter two get the sign wrong due to losing the phase information. It is perhaps worth noting that the latter two (`xrft.fft` and `npft.fft`) require the amplitudes to be multiplied by dx to be consistent with theory while `xrft.dft` automatically takes care of this with the flag `true_amplitude=True`:

$$\mathcal{F}(da)(f) = \int_{-\infty}^{+\infty} da(x) e^{-2\pi i f x} dx \rightarrow \text{xrft.dft}(da)(f[m]) = \sum_n da(x[n]) e^{-2\pi i f[m] x[n]} \Delta x$$

Perform the inverse transform

[5]:
```
ida_dft = xrft.idft(da_dft, true_phase=True, true_amplitude=True) # Signal in direct␣
↪space
ida_fft = xrft.ifft(da_fft)
```

[19]:
```
fig, ax = plt.subplots(figsize=(12,4))
fig.set_tight_layout(True)
ida_dft.real.plot(ax=ax, linestyle='-', c='k', lw=4, label='phase preservation')
ax.plot(x, ida_fft.real, linestyle='', marker='+', label='no phase preservation', alpha=.
↪6) # w/out the phase information, the coordinates are lost
da.plot(ax=ax, ls='--', lw=3, label='original signal')
ax.plot(x, npft.ifft(da_npft).real, ls=':', label='inverse of numpy fft')
ax.set_xlim([-8,8])
ax.legend(loc='upper left');
```



Although `xrft.ifft` misses the amplitude scaling (viz. resolution in wavenumber or frequency), since it is the inverse of the Fourier transform uncorrected for dx, the result becomes consistent with `xrft.idft`. In other words, `xrft.fft` (and `npft.fft`) misses the dx scaling and `xrft.ifft` (and `npft.ifft`) misses the df $(= 1/(N \times dx))$ scaling. When applying the two operators in conjuction by doing `ifft(fft())`, there is a $1/N$ $(= dx \times df)$ factor missing which is, in fact, *arbitrarily included in the ``ifft`` definition as a normalization factor <https://numpy.org/doc/stable/reference/routines.fft.html#module-numpy.fft>`__*. By incorporating the right scalings in `xrft.dft` and `xrft.idft`, there is no more consideration of the number of data points (N):

$$\mathcal{F}^{-1}(\mathcal{F}(da))(x) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \mathcal{F}(da)(f)e^{2\pi i f x}df \rightarrow \text{xrft.idft}(\text{xrft.dft}(da))(x[n]) = \sum_m \text{xrft.dft}(da)(f[m])e^{2\pi i f[m]x[n]}\Delta f$$

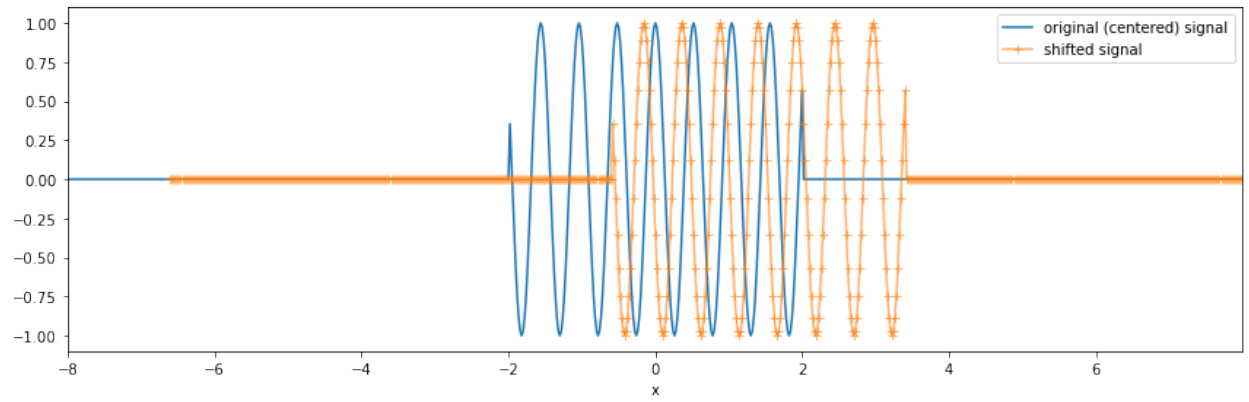
Synthetic data not centered around zero

Now let's shift the coordinates so that they are not centered.

This is where the ``xrft`` magic happens. With the relevant flags, `xrft`'s dft can preserve information about the data's location in its original space. This information is not preserved in a `numpy` fourier transform. This section demonstrates how to preserve this information using the `true_phase=True`, `true_amplitude=True` flags.

[26]:
```
nshift = 70                             # defining a shift
x0 = dx*nshift
nda = da.shift(x=nshift).dropna('x')
```

[27]:
```
fig, ax = plt.subplots(figsize=(12,4))
fig.set_tight_layout(True)
da.plot(ax=ax, label='original (centered) signal')
nda.plot(ax=ax, marker='+', label='shifted signal', alpha=.6)
ax.set_xlim([-8,nda.x.max()])
ax.legend();
```

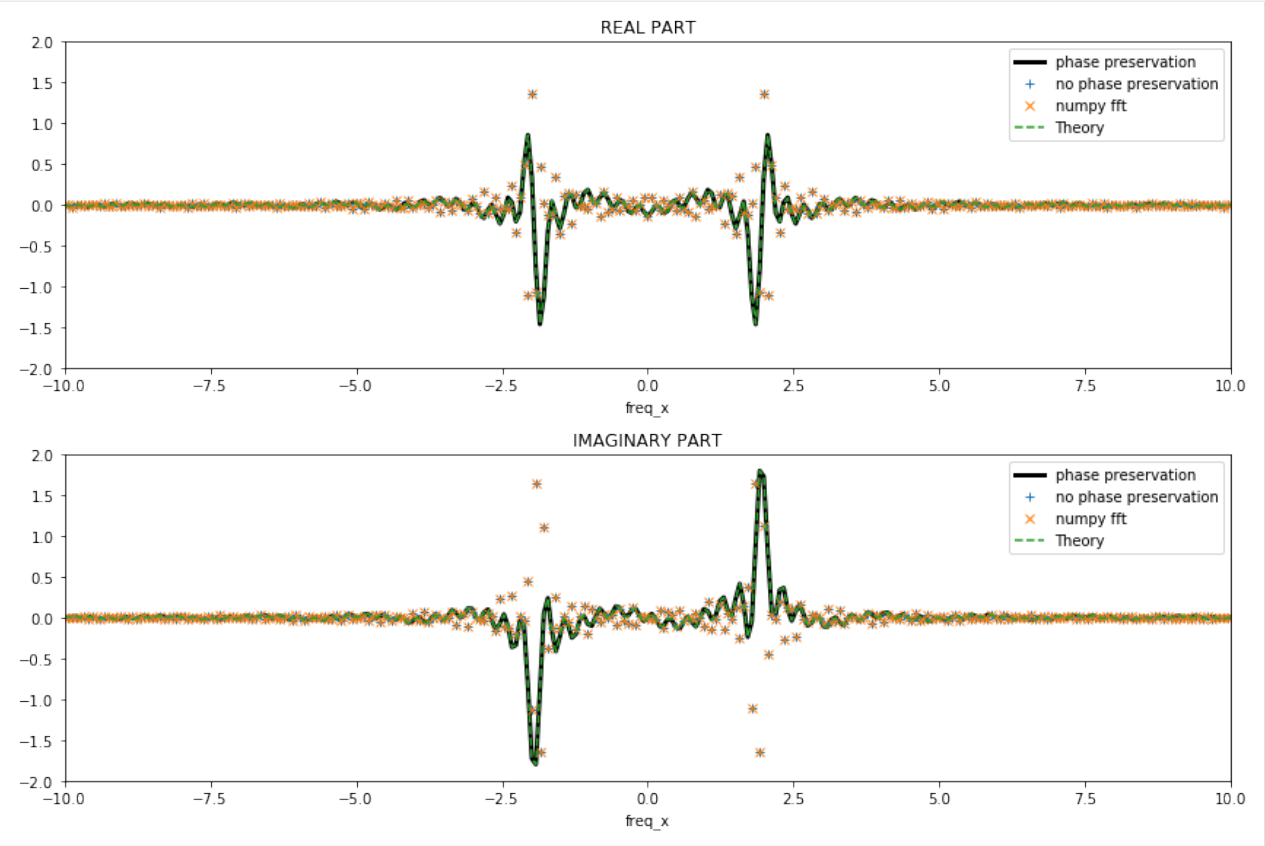


We consider again the Fourier transform.

[28]:
```
nda_dft = xrft.dft(nda, true_phase=True, true_amplitude=True) # Fourier Transform w/␣
↪phase preservation
nda_fft = xrft.fft(nda)                                       # Fourier Transform w/out␣
↪phase preservation
nda_npft = npft.fft(nda)
```

[29]:
```
nk = nda_dft.freq_x # wavenumber axis
TF_ns = T/2*(np.sinc(T*(nk-k0)) + np.sinc(T*(nk+k0)))*np.exp(-2j*np.pi*nk*x0) #␣
↪Theoretical FT (Note the additional phase)
```

[30]:
```
fig, (ax1,ax2) = plt.subplots(figsize=(12,8), nrows=2, ncols=1)
fig.set_tight_layout(True)

(nda_dft.real).plot(ax=ax1, linestyle='-', lw=3, c='k', label='phase preservation')
((nda_fft*dx).real).plot(ax=ax1, linestyle='', marker='+',label='no phase preservation')
ax1.plot(nk, (npft.fftshift(nda_npft)*dx).real, linestyle='', marker='x',label='numpy fft
↪')
ax1.plot(nk, TF_ns.real, linestyle='--', label='Theory')
ax1.set_xlim([-10,10])
ax1.set_ylim([-2.,2])
ax1.legend()
ax1.set_title('REAL PART')

(nda_dft.imag).plot(ax=ax2, linestyle='-', lw=3, c='k', label='phase preservation')
((nda_fft*dx).imag).plot(ax=ax2, linestyle='', marker='+', label='no phase preservation')
ax2.plot(nk, (npft.fftshift(nda_npft)*dx).imag, linestyle='', marker='x',label='numpy fft
↪')
ax2.plot(nk, TF_ns.imag, linestyle='--', label='Theory')
ax2.set_xlim([-10,10])
ax2.set_ylim([-2.,2.])
ax2.legend()
ax2.set_title('IMAGINARY PART');
```

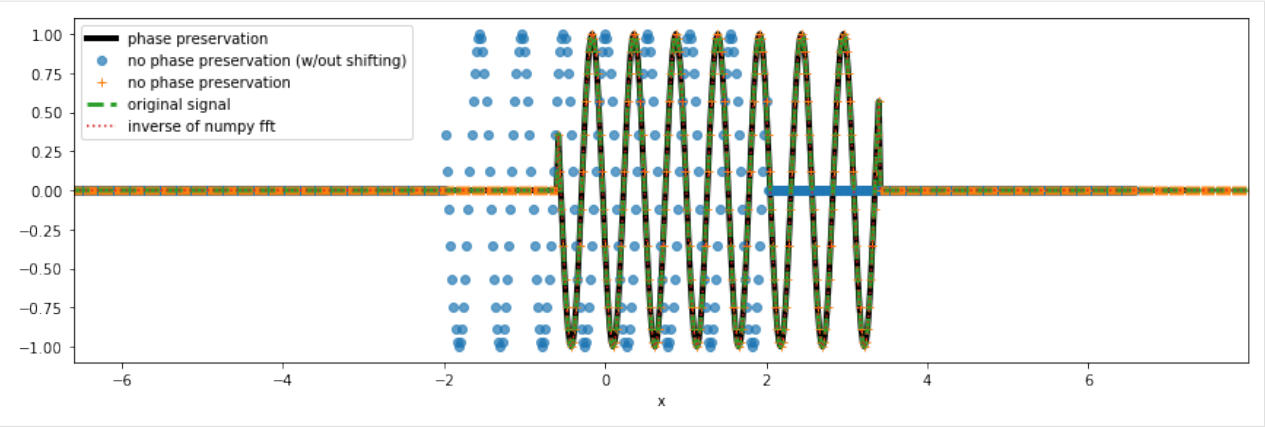


The expected additional phase (i.e. the complex term; $e^{-i2\pi kx_0}$) that appears in theory is retrieved with `xrft.dft` but not with `xrft.fft` nor `npft.fft`. This is because in `npft.fft`, the input data is expected to be centered around zero. **In the current version of ``xrft``, the behavior of ``xrft.dft`` defaults to ``xrft.fft`` so set the flags ``true_phase=True`` and ``true_amplitude=True`` in order to have the results matching with theory.**

Now, let's take the inverse transform.

[31]:
```
inda_dft = xrft.idft(nda_dft, true_phase=True, true_amplitude=True) # Signal in direct␣
↪space
inda_fft = xrft.ifft(nda_fft)
```

[39]:
```
fig, ax = plt.subplots(figsize=(12,4))
fig.set_tight_layout(True)
inda_dft.real.plot(ax=ax, linestyle='-', c='k', lw=4, label='phase preservation')
ax.plot(x[:len(inda_fft.real)], inda_fft.real, linestyle='', marker='o', alpha=.7,
        label='no phase preservation (w/out shifting)')
ax.plot(x[nshift:], inda_fft.real, linestyle='', marker='+', label='no phase preservation
↪')
nda.plot(ax=ax, ls='--', lw=3, label='original signal')
ax.plot(x[nshift:], npft.ifft(nda_npft).real, ls=':', label='inverse of numpy fft')
ax.set_xlim([nda.x.min(),nda.x.max()])
ax.legend(loc='upper left');
```



Note that we are only able to match the inverse transforms of `xrft.ifft` and `npft.ifft` to the data `nda` to it being Fourier transformed because we "know" the original data `da` was shifted by `nshift` datapoints as we see in `x[nshift:]` (compare the blue dots and orange crosses where without the knowledge of the shift, we may assume that the data were centered around zero). **Using ``xrft.idft`` along with ``xrft.dft`` with the flags ``true_phase=True`` and ``true_amplitude=True`` automatically takes care of the information of shifted coordinates.**

1.5.2 A case with real data

Load atmosheric temperature from the NMC reanalysis.

[4]:
```
da = xr.tutorial.open_dataset("air_temperature").air
da
```

[4]:
```
<xarray.DataArray 'air' (time: 2920, lat: 25, lon: 53)>
[3869000 values with dtype=float32]
Coordinates:
  * lat      (lat) float32 75.0 72.5 70.0 67.5 65.0 ... 25.0 22.5 20.0 17.5 15.0
  * lon      (lon) float32 200.0 202.5 205.0 207.5 ... 322.5 325.0 327.5 330.0
  * time     (time) datetime64[ns] 2013-01-01 ... 2014-12-31T18:00:00
Attributes:
    long_name:     4xDaily Air temperature at sigma level 995
    units:         degK
    precision:     2
    GRIB_id:       11
    GRIB_name:     TMP
    var_desc:      Air temperature
    dataset:       NMC Reanalysis
    level_desc:    Surface
    statistic:     Individual Obs
    parent_stat:   Other
    actual_range:  [185.16 322.1 ]
```

[6]:
```
Fda = xrft.dft(da.isel(time=0), dim="lat", true_phase=True, true_amplitude=True)
Fda
```

[6]:
```
<xarray.DataArray (freq_lat: 25, lon: 53)>
array([[ 55.72721061-46.86182115j,  54.88410513-45.81648436j,
         54.44861105-45.4792758j , ...,  63.78368643-52.78354988j,
         60.99712143-50.28091047j,  57.94756055-48.51852927j],
```

(continues on next page)

(continued from previous page)

```
       [-38.90906614-66.9663849j , -38.90038095-67.92497252j,
        -38.544492  -68.17925905j, ..., -41.94737401-74.09175983j,
        -40.00936528-70.47655747j, -39.05651073-68.04032158j],
       [-77.82019891+12.50876021j, -79.02288653+10.06164636j,
        -80.34175059 +9.81548668j, ..., -86.83039434+26.4819109j ,
        -84.02694401+23.31755583j, -81.26451369+21.4268003j ],
       ...,
       [-77.82019891-12.50876021j, -79.02288653-10.06164636j,
        -80.34175059 -9.81548668j, ..., -86.83039434-26.4819109j ,
        -84.02694401-23.31755583j, -81.26451369-21.4268003j ],
       [-38.90906614+66.9663849j , -38.90038095+67.92497252j,
        -38.544492  +68.17925905j, ..., -41.94737401+74.09175983j,
        -40.00936528+70.47655747j, -39.05651073+68.04032158j],
       [ 55.72721061+46.86182115j,  54.88410513+45.81648436j,
         54.44861105+45.4792758j , ...,  63.78368643+52.78354988j,
         60.99712143+50.28091047j,  57.94756055+48.51852927j]])
Coordinates:
  * lon       (lon) float32 200.0 202.5 205.0 207.5 ... 322.5 325.0 327.5 330.0
    time      datetime64[ns] 2013-01-01
  * freq_lat  (freq_lat) float64 -0.192 -0.176 -0.16 -0.144 ... 0.16 0.176 0.192
```

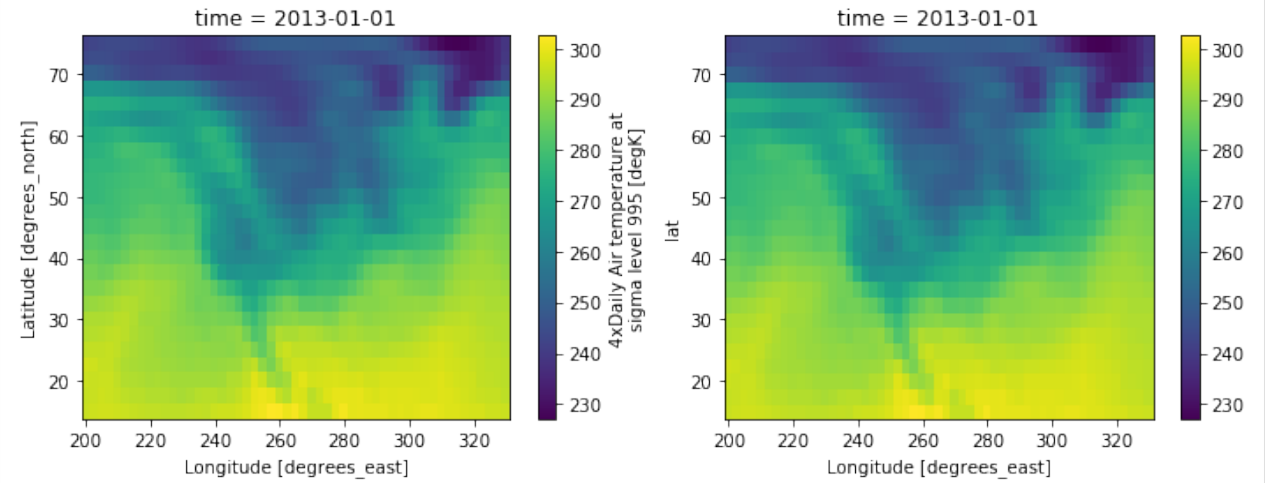
The coordinate metadata is lost during the DFT (or any Fourier transform) operation so we need to specify the `lag` to retrieve the latitudes back in the inverse transform. The original latitudes are centered around 45° so we set the lag to `lag=45`.

[8]:
```
Fda_1 = xrft.idft(Fda, dim="freq_lat", true_phase=True, true_amplitude=True, lag=45)
Fda_1
```

[8]:
```
<xarray.DataArray (lat: 25, lon: 53)>
array([[296.29000854-7.01488818e-16j, 296.79000854-2.41028295e-15j,
        297.1000061 -1.08051561e-15j, ..., 296.8999939 +2.07870428e-15j,
        296.79000854+1.39068454e-15j, 296.6000061 +1.98243140e-15j],
       [295.8999939 -1.36617001e-16j, 296.19998169-3.08854147e-15j,
        296.79000854-2.27797690e-16j, ..., 295.8999939 -2.06120304e-15j,
        295.8999939 -7.63307736e-16j, 295.19998169+2.90958929e-15j],
       [296.6000061 +2.18513309e-15j, 296.19998169-5.34587573e-16j,
        296.3999939 -1.70159409e-15j, ..., 295.3999939 -9.67078004e-16j,
        295.1000061 -2.97325892e-15j, 294.69998169+2.84108954e-15j],
       ...,
       [250.        +1.32015223e-15j, 249.79998779+5.34587573e-16j,
        248.88999939-1.80369123e-15j, ..., 233.19999695+9.67078004e-16j,
        236.38999939+3.00722368e-17j, 241.69999695+3.04528382e-15j],
       [243.79998779-1.32169378e-16j, 244.5       -4.76080394e-15j,
        244.69999695-1.17678849e-15j, ..., 232.79998779+1.86297913e-15j,
        235.29998779-3.02882224e-16j, 239.29998779-9.67078004e-16j],
       [241.19999695+5.26510200e-16j, 242.5       -1.34198513e-15j,
        243.5       -3.83146461e-16j, ..., 232.79998779+2.46618241e-15j,
        235.5       +3.62856196e-16j, 238.59999084+4.17030299e-16j]])
Coordinates:
  * lon      (lon) float32 200.0 202.5 205.0 207.5 ... 322.5 325.0 327.5 330.0
    time     datetime64[ns] 2013-01-01
  * lat      (lat) float64 15.0 17.5 20.0 22.5 25.0 ... 65.0 67.5 70.0 72.5 75.0
```

[20]:
```
fig, (ax1,ax2) = plt.subplots(figsize=(12,4), nrows=1, ncols=2)
da.isel(time=0).plot(ax=ax1)
Fda_1.real.plot(ax=ax2)
```

[20]:
```
<matplotlib.collections.QuadMesh at 0x1226607f0>
```



We see the inverse DFT of the Fourier transformed original temperature data returns the original data.

[ ]:
```
```

1.6 Example of Parseval's theorem

[1]:
```
import numpy as np
import numpy.testing as npt
import xarray as xr
import xrft
import numpy.fft as npft
import dask.array as dsar
import matplotlib.pyplot as plt
%matplotlib inline
```

First, we show that ``xrft.dft`` satisfies the Parseval's theorem exactly for a non-windowed signal

For one-dimensional data:

$$\sum_x (da)^2 \Delta x = \sum_k \mathcal{F}(da)[\mathcal{F}(da)]^* \Delta k.$$

Generate synthetic data

[10]:
```
Nx = 40
dx = np.random.rand()
da = xr.DataArray(
        np.random.rand(Nx) + 1j * np.random.rand(Nx),
        dims="x",
        coords={"x": dx * (np.arange(-Nx // 2, -Nx // 2 + Nx)
                           + np.random.randint(-Nx // 2, Nx // 2)
```

(continues on next page)

(continued from previous page)

```
                        )
            },
              )
```

[2]:
```
###############
# Assert Parseval's using xrft.dft
###############
FT = xrft.dft(da, dim="x", true_phase=True, true_amplitude=True)
npt.assert_almost_equal(
        (np.abs(da) ** 2).sum() * dx, (np.abs(FT) ** 2).sum() * FT["freq_x"].spacing
                        )

###############
# Assert Parseval's using xrft.power_spectrum with scaling='density'
###############
ps = xrft.power_spectrum(da, dim="x")
npt.assert_almost_equal(
        ps.sum(),
        (np.abs(da) ** 2).sum() * dx
                        )
```

For two-dimensional data:

$$\sum_x \sum_y (da)^2 \Delta x \Delta y = \sum_k \sum_l \mathcal{F}(da)\mathcal{F}(da)^* \Delta k \Delta l.$$

Generate synthetic data

[11]:
```
Ny = 60
dx, dy = (np.random.rand(), np.random.rand())
da2 = xr.DataArray(
        np.random.rand(Nx, Ny) + 1j * np.random.rand(Nx, Ny),
        dims=["x", "y"],
        coords={"x": dx
                * (
                    np.arange(-Nx // 2, -Nx // 2 + Nx)
                    + np.random.randint(-Nx // 2, Nx // 2)
                ),
                "y": dy
                * (
                    np.arange(-Ny // 2, -Ny // 2 + Ny)
                    + np.random.randint(-Ny // 2, Ny // 2)
                ),
               },
                   )
```

[3]:
```
###############
# Assert Parseval's using xrft.dft
###############
FT2 = xrft.dft(da2, dim=["x", "y"], true_phase=True, true_amplitude=True)
npt.assert_almost_equal(
        (np.abs(FT2) ** 2).sum() * FT2["freq_x"].spacing * FT2["freq_y"].spacing,
```

(continues on next page)

(continued from previous page)

```
        (np.abs(da2) ** 2).sum() * dx * dy,
                        )

###############
# Assert Parseval's using xrft.power_spectrum with scaling='density'
###############
ps2 = xrft.power_spectrum(da2, dim=["x", "y"])
npt.assert_almost_equal(
        ps2.sum(),
        (np.abs(da2) ** 2).sum() * dx * dy
                        )

###############
# Assert Parseval's using xrft.power_spectrum with scaling='spectrum'
###############
ps2 = xrft.power_spectrum(da2, dim=["x", "y"], scaling='spectrum')
npt.assert_almost_equal(
        ps2.sum() / (ps2.freq_x.spacing * ps2.freq_y.spacing),
        (np.abs(da2) ** 2).sum() * dx * dy,
                        )
```

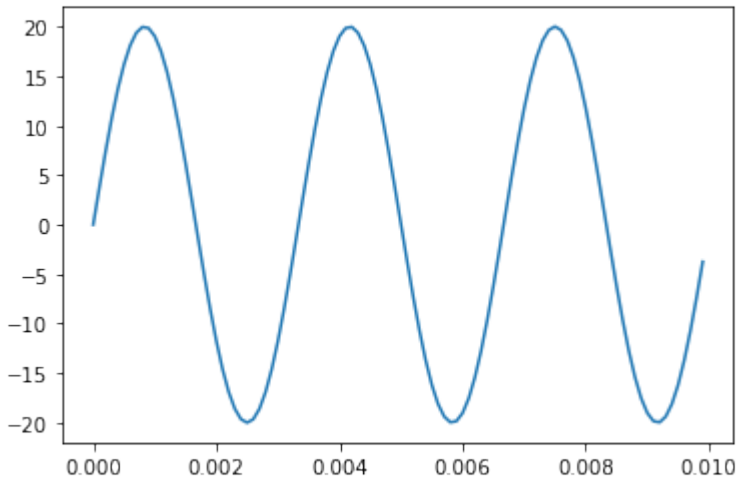
Now, we show how Parseval's theorem is approximately satisfied for windowed data

$$\frac{1}{\langle w^2 \rangle} \sum_x (w\ da)^2 \Delta x = \sum_x (da)^2 \Delta x \approx \frac{1}{\langle w^2 \rangle} \sum_k \mathcal{F}(w) \circ \mathcal{F}(da) [\mathcal{F}(w) \circ \mathcal{F}(da)]^* \Delta k,$$

where w is the windowing function, $\circ$ is the convolution operator and $\langle \cdot \rangle$ is the area sum, namely, $\sum_x$.

Generate synthetic data: A 300Hz sine wave with an RMS^2 of 200.

[3]:
```
A = 20
fs = 1e4
n_segments = int(fs // 10)
fsig = 300
ii = int(fsig * n_segments // fs)  # frequency index of fsig

tt = np.arange(fs) / fs
x = A * np.sin(2 * np.pi * fsig * tt)
plt.plot(tt[:100],x[:100]);
```



Assert Parseval's for different windowing functions

Depending on the `scaling` flag, a different correction is applied to the windowed spectrum:

- `scaling='density'`: **Energy correction** - this corrects for the energy (integral) of the spectrum. It is typically applied to the power spectral density (including cross power spectral density) and rescales the spectrum by `1.0 / (window**2).mean()`. It ensures that the integral of the spectral density (approximately) matches the RMS^2 of the signal (i.e. that Parseval's theorem is satisfied).
- `scaling='spectrum'`: **Amplitude correction** - this corrects the amplitude of peaks in the spectrum and rescales the spectrum by `1.0 / window.mean()**2`. It is typically applied to the power spectrum (i.e. not density) and is most useful in strongly periodic signals. It ensures, for example, that the peak in the power spectrum of a 300 Hz sine wave with $\text{RMS}^2 = 200$ has a magnitude of 200.

These scalings replicate the default behaviour of scipy spectral functions like `scipy.signal.periodogram` and `scipy.signal.welch` (cf. Section 11.5.2. of Bendat & Piersol, 2011; Section 10.3 of Brandt, 2011).

[4]:
```
RMS = np.sqrt(np.mean((x**2)))

windows = np.array([["hann","bartlett"],["tukey","flattop"]])
```

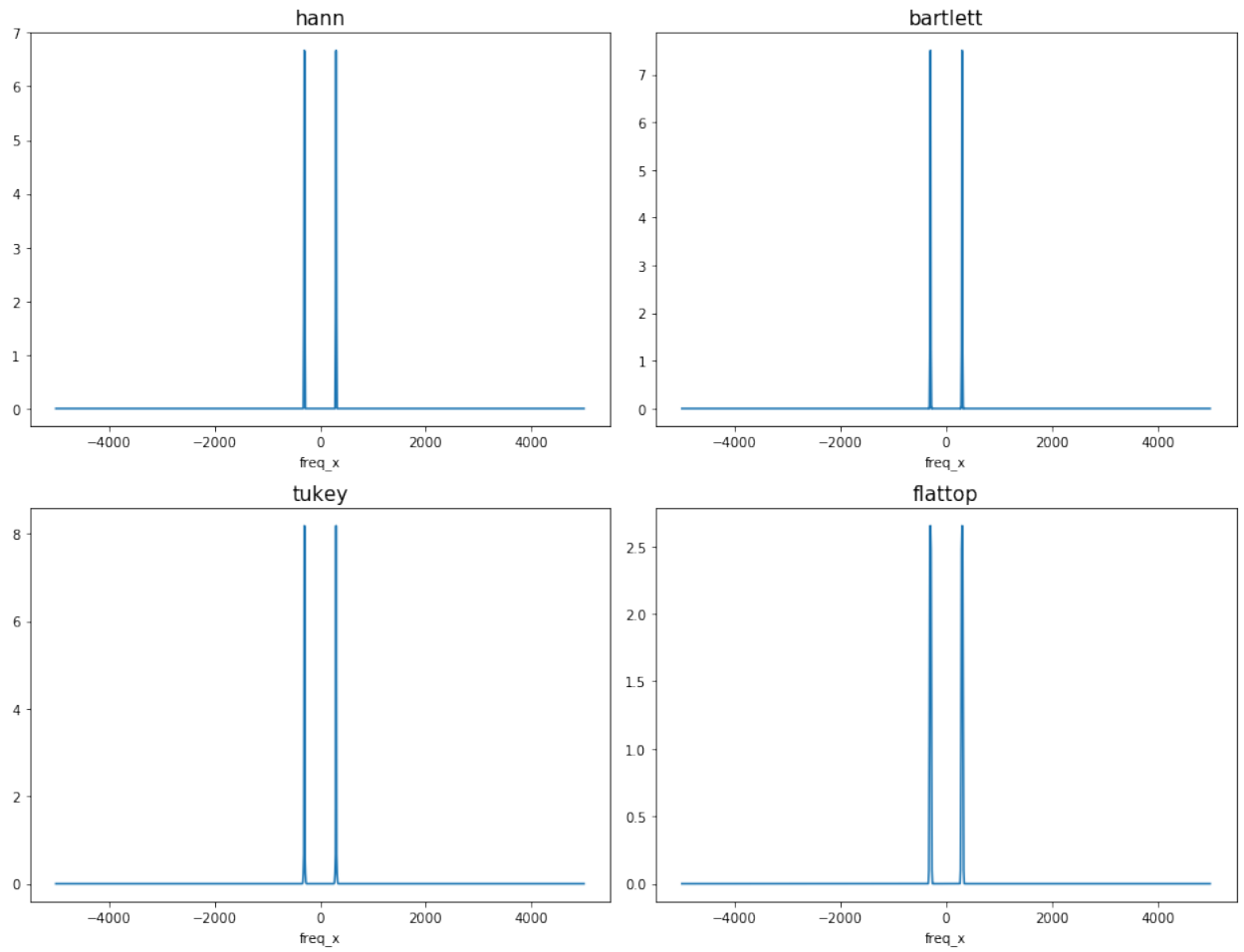
Check the energy correction for ``scaling='density'``: With `window_correction=True`, the spectrum integrates to RMS^2.

[5]:
```
fig, axes = plt.subplots(figsize=(13,10), nrows=2, ncols=2)
fig.set_tight_layout(True)

for window_type in windows.ravel():

    x_da = xr.DataArray(x, coords=[tt], dims=["x"]).chunk({"x": n_segments})

    ps = xrft.power_spectrum(
        x_da,
        dim="x",
        window=window_type,
        chunks_to_segments=True,
        window_correction=True,
    ).mean("x_segment")
```

(continues on next page)

(continued from previous page)

```
    ps.plot(ax=axes[np.where(windows==window_type)[0][0],np.where(windows==window_
↪type)[1][0]])
    axes[np.where(windows==window_type)[0][0],np.where(windows==window_type)[1][0]].set_
↪title(window_type, fontsize=15)

    npt.assert_allclose(
        np.trapz(ps.values, ps.freq_x.values),
        RMS**2,
        rtol=1e-3
    )
```



The maximum amplitude differs amongst cases with different windows due to the difference in noise floor.

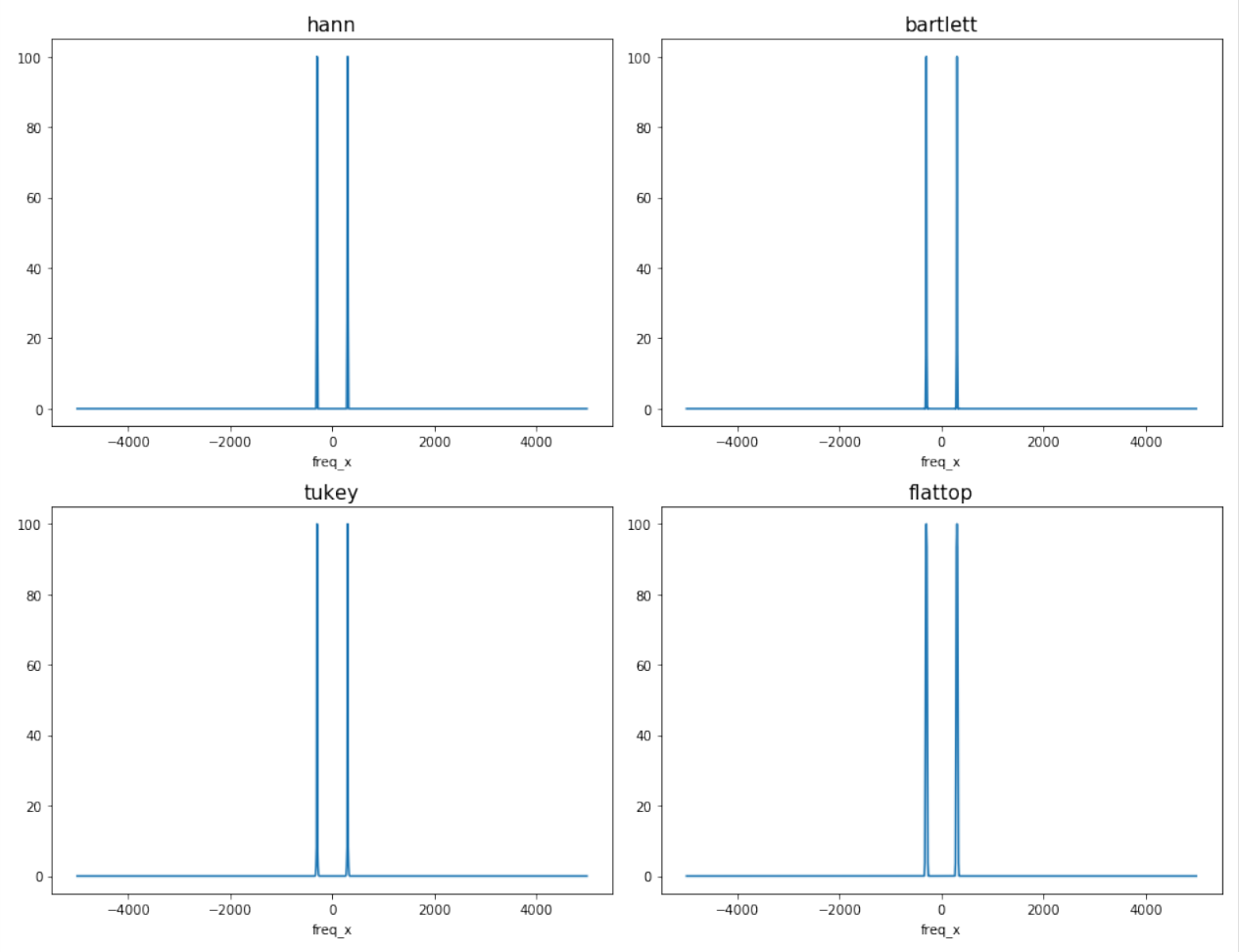
Check the amplitude correction for ``scaling='spectrum'``: With `window_correction=True`, the peak of the two-sided spectrum has a magnitude of $\mathrm{RMS}^2/2$.

[6]:
```
fig, axes = plt.subplots(figsize=(13,10), nrows=2, ncols=2)
fig.set_tight_layout(True)

for window_type in windows.ravel():
```

(continues on next page)

(continued from previous page)

```
    x_da = xr.DataArray(x, coords=[tt], dims=["x"]).chunk({"x": n_segments})

    ps = xrft.power_spectrum(
        x_da,
        dim="x",
        window=window_type,
        chunks_to_segments=True,
        scaling="spectrum",
        window_correction=True,
    ).mean("x_segment")

    ps.plot(ax=axes[np.where(windows==window_type)[0][0],np.where(windows==window_
↪type)[1][0]])
    axes[np.where(windows==window_type)[0][0],np.where(windows==window_type)[1][0]].set_
↪title(window_type, fontsize=15)

    # The factor of 0.5 is there because we're checking the two-sided spectrum
    npt.assert_allclose(ps.sel(freq_x=fsig), 0.5 * RMS**2)
```



The maximum amplitudes are now all the same amongst different windows applied with the **amplitude correction**.

[ ]:

[1]:
```
import numpy as np
import numpy.testing as npt
import xarray as xr
import xrft
import dask.array as dsar
from matplotlib import colors
import matplotlib.pyplot as plt
%matplotlib inline
```

1.7 Parallelized Bartlett's Method

For long data sets that have reached statistical equilibrium, it is useful to chunk the data, calculate the periodogram for each chunk and then take the average to reduce variance.

[2]:
```
n = int(2**8)
da = xr.DataArray(np.random.rand(n,int(n/2),int(n/2)), dims=['time','y','x'])
da
```

[2]:
```
<xarray.DataArray (time: 256, y: 128, x: 128)>
array([[[ 0.493341,  0.28303 , ...,  0.434256,  0.616031],
        [ 0.777314,  0.629644, ...,  0.152931,  0.445424],
        ...,
        [ 0.562456,  0.022227, ...,  0.88538 ,  0.054687],
        [ 0.381456,  0.908454, ...,  0.843443,  0.706326]],

       [[ 0.469143,  0.241104, ...,  0.249369,  0.830898],
        [ 0.283305,  0.438634, ...,  0.893666,  0.242556],
        ...,
        [ 0.897823,  0.187038, ...,  0.977466,  0.270899],
        [ 0.252733,  0.425873, ...,  0.228847,  0.954393]],

       ...,
       [[ 0.936424,  0.793693, ...,  0.406293,  0.272336],
        [ 0.917752,  0.83908 , ...,  0.954489,  0.151129],
        ...,
        [ 0.081756,  0.016332, ...,  0.524886,  0.87095 ],
        [ 0.677224,  0.41488 , ...,  0.12199 ,  0.689685]],

       [[ 0.193302,  0.113419, ...,  0.083486,  0.784332],
        [ 0.695728,  0.376776, ...,  0.278004,  0.026373],
        ...,
        [ 0.677775,  0.255296, ...,  0.112851,  0.46325 ],
        [ 0.598086,  0.529324, ...,  0.267431,  0.65419 ]]])
Dimensions without coordinates: time, y, x
```

1.7.1 One dimension

Discrete Fourier Transform

[3]:
```
daft = xrft.dft(da.chunk({'time':int(n/4)}), dim=['time'], shift=False , chunks_to_
↪segments=True).compute()
daft
```

[3]:
```
<xarray.DataArray 'fftn-917988d0ce7d7da01b5f7a3cf2bb9a26' (time_segment: 4, freq_time:␣
↪64, y: 128, x: 128)>
array([[[[ 30.737014+0.j      , ...,  31.659135+0.j      ],
         ...,
         [ 31.308938+0.j      , ...,  31.768846+0.j      ]],

        ...,
        [[  1.928097-0.118076j, ...,   0.732440+2.07656j ],
         ...,
         [  0.225814+1.256083j, ...,   0.244113-1.276807j]]],


       ...,
       [[[ 37.777908+0.j      , ...,  30.996848+0.j      ],
         ...,
         [ 28.650088+0.j      , ...,  35.362874+0.j      ]],

        ...,
        [[ -1.780642+0.477772j, ...,   2.575858+1.71943j ],
         ...,
         [  3.149759-2.664934j, ...,   1.872009-2.977565j]]]])
Coordinates:
  * time_segment       (time_segment) int64 0 1 2 3
  * freq_time          (freq_time) float64 0.0 0.01562 0.03125 0.04688 ...
  * y                  (y) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ...
  * x                  (x) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ...
    freq_time_spacing  float64 0.01562
```

[4]:
```
data = da.chunk({'time':int(n/4)}).data
data_rs = data.reshape((4,int(n/4),int(n/2),int(n/2)))
da_rs = xr.DataArray(data_rs, dims=['time_segment','time','y','x'])
da1 = xr.DataArray(dsar.fft.fftn(data_rs, axes=[1]).compute(),
                   dims=['time_segment','freq_time','y','x'])
da1
```

[4]:
```
<xarray.DataArray (time_segment: 4, freq_time: 64, y: 128, x: 128)>
array([[[[ 30.737014+0.j      , ...,  31.659135+0.j      ],
         ...,
         [ 31.308938+0.j      , ...,  31.768846+0.j      ]],

        ...,
        [[  1.928097-0.118076j, ...,   0.732440+2.07656j ],
         ...,
         [  0.225814+1.256083j, ...,   0.244113-1.276807j]]],
```

(continues on next page)

(continued from previous page)

```
       ...,
       [[[ 37.777908+0.j      , ...,  30.996848+0.j      ],
         ...,
         [ 28.650088+0.j      , ...,  35.362874+0.j      ]],

        ...,
        [[ -1.780642+0.477772j, ...,   2.575858+1.71943j ],
         ...,
         [  3.149759-2.664934j, ...,   1.872009-2.977565j]]]])
Dimensions without coordinates: time_segment, freq_time, y, x
```

We assert that our calculations give equal results.

[5]:
```
npt.assert_almost_equal(da1, daft.values)
```

Power Spectrum

[6]:
```
ps = xrft.power_spectrum(da.chunk({'time':int(n/4)}), dim=['time'], chunks_to_
↪segments=True)
ps
```

[6]:
```
<xarray.DataArray 'concatenate-183433100cd82e429170a4fe2f9c4cbb' (time_segment: 4, freq_
↪time: 64, y: 128, x: 128)>
dask.array<truediv, shape=(4, 64, 128, 128), dtype=float64, chunksize=(1, 32, 128, 128)>
Coordinates:
  * time_segment       (time_segment) int64 0 1 2 3
  * freq_time          (freq_time) float64 -0.5 -0.4844 -0.4688 -0.4531 ...
  * y                  (y) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ...
  * x                  (x) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ...
    freq_time_spacing  float64 0.01562
```

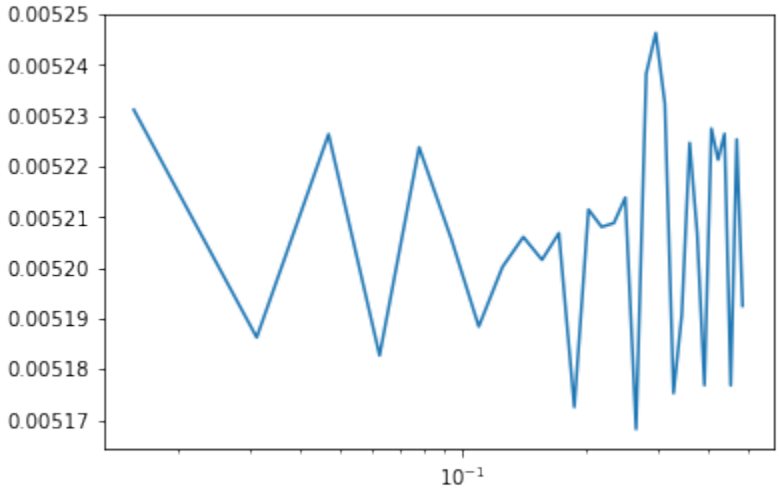
Taking the mean over the segments gives the Barlett's estimate.

[7]:
```
ps = ps.mean(['time_segment','y','x'])
ps
```

[7]:
```
<xarray.DataArray 'concatenate-183433100cd82e429170a4fe2f9c4cbb' (freq_time: 64)>
dask.array<mean_agg-aggregate, shape=(64,), dtype=float64, chunksize=(32,)>
Coordinates:
  * freq_time          (freq_time) float64 -0.5 -0.4844 -0.4688 -0.4531 ...
    freq_time_spacing  float64 0.01562
```

[8]:
```
fig, ax = plt.subplots()
ax.semilogx(ps.freq_time[int(n/8)+1:], ps[int(n/8)+1:])
```

[8]:
```
[<matplotlib.lines.Line2D at 0x10ebc9518>]
```



1.7.2 Two dimension

Discrete Fourier Transform

[9]:
```
daft = xrft.dft(da.chunk({'y':32,'x':32}), dim=['y','x'], shift=False , chunks_to_
↪segments=True).compute()
daft
```

[9]:
```
<xarray.DataArray 'fftn-8077935acd6b48b40d6593c688c326b2' (time: 256, y_segment: 4, freq_
↪y: 32, x_segment: 4, freq_x: 32)>
array([[[[[ 505.090962 +0.j      , ...,    3.673241 +2.033024j],
          ...,
          [ 506.979486 +0.j      , ...,    2.672219 +8.645102j]],

         ...,
         [[  -1.746757 -1.347122j, ...,   -2.183099+17.472835j],
          ...,
          [   3.450049 +3.832201j, ...,   -4.072164 -7.279733j]]],


        ...,
        [[[ 504.971751 +0.j      , ...,   -6.610465-12.385931j],
          ...,
          [ 512.756185 +0.j      , ...,   -4.344255 -8.458134j]],

         ...,
         [[  -7.979198 -7.454325j, ...,   -2.962019 +6.43059j ],
          ...,
          [   4.024805 +3.72519j , ...,   -8.242673 -8.259182j]]]],


       ...,
       [[[[ 518.573138 +0.j      , ...,    0.573928-10.006888j],
```

(continues on next page)

(continued from previous page)

```
          ...,
          [ 520.423164 +0.j      , ...,   -1.110088 +0.141936j]],

         ...,
         [[   2.043005 -3.116515j, ...,    8.697924 -5.116488j],
          ...,
          [   3.702009 -7.202762j, ...,  -12.007770 +3.514272j]]],


        ...,
        [[[ 523.615806 +0.j      , ...,   -9.301065 +4.935474j],
          ...,
          [ 521.535950 +0.j      , ...,    6.826755 +1.688166j]],

         ...,
         [[   2.157400-14.676636j, ...,   -1.865237-11.408717j],
          ...,
          [   0.651302 +0.531716j, ...,    5.861882 +5.968681j]]]])
Coordinates:
  * time            (time) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 ...
  * y_segment       (y_segment) int64 0 1 2 3
  * freq_y          (freq_y) float64 0.0 0.03125 0.0625 0.09375 0.125 0.1562 ...
  * x_segment       (x_segment) int64 0 1 2 3
  * freq_x          (freq_x) float64 0.0 0.03125 0.0625 0.09375 0.125 0.1562 ...
    freq_y_spacing  float64 0.03125
    freq_x_spacing  float64 0.03125
```

[10]:
```
data = da.chunk({'y':32,'x':32}).data
data_rs = data.reshape((256,4,32,4,32))
da_rs = xr.DataArray(data_rs, dims=['time','y_segment','y','x_segment','x'])
da2 = xr.DataArray(dsar.fft.fftn(data_rs, axes=[2,4]).compute(),
                   dims=['time','y_segment','freq_y','x_segment','freq_x'])
da2
```

[10]:
```
<xarray.DataArray (time: 256, y_segment: 4, freq_y: 32, x_segment: 4, freq_x: 32)>
array([[[[[ 505.090962 +0.j      , ...,    3.673241 +2.033024j],
          ...,
          [ 506.979486 +0.j      , ...,    2.672219 +8.645102j]],

         ...,
         [[  -1.746757 -1.347122j, ...,   -2.183099+17.472835j],
          ...,
          [   3.450049 +3.832201j, ...,   -4.072164 -7.279733j]]],


        ...,
        [[[ 504.971751 +0.j      , ...,   -6.610465-12.385931j],
          ...,
          [ 512.756185 +0.j      , ...,   -4.344255 -8.458134j]],

         ...,
         [[  -7.979198 -7.454325j, ...,   -2.962019 +6.43059j ],
```

(continues on next page)

(continued from previous page)

```
          ...,
          [   4.024805 +3.72519j , ...,   -8.242673 -8.259182j]]]],



       ...,
       [[[[ 518.573138 +0.j      , ...,    0.573928-10.006888j],
          ...,
          [ 520.423164 +0.j      , ...,   -1.110088 +0.141936j]],

         ...,
         [[   2.043005 -3.116515j, ...,    8.697924 -5.116488j],
          ...,
          [   3.702009 -7.202762j, ...,  -12.007770 +3.514272j]]],


        ...,
        [[[ 523.615806 +0.j      , ...,   -9.301065 +4.935474j],
          ...,
          [ 521.535950 +0.j      , ...,    6.826755 +1.688166j]],

         ...,
         [[   2.157400-14.676636j, ...,   -1.865237-11.408717j],
          ...,
          [   0.651302 +0.531716j, ...,    5.861882 +5.968681j]]]]])
Dimensions without coordinates: time, y_segment, freq_y, x_segment, freq_x
```

We assert that our calculations give equal results.

[11]:
```
npt.assert_almost_equal(da2, daft.values)
```

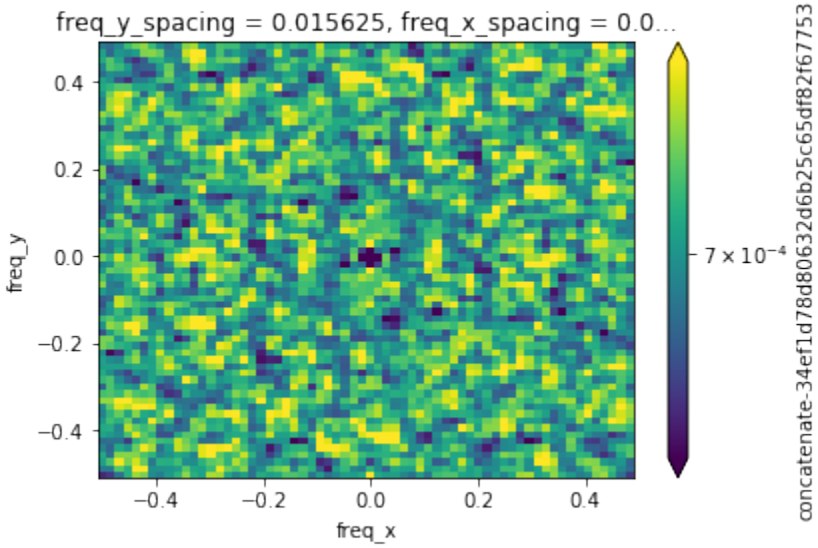
Power Spectrum

[14]:
```
ps = xrft.power_spectrum(da.chunk({'time':1,'y':64,'x':64}), dim=['y','x'],
                         chunks_to_segments=True, window='True', detrend='linear')
ps = ps.mean(['time','y_segment','x_segment'])
ps
```

[14]:
```
<xarray.DataArray 'concatenate-34ef1d78d80632d6b25c65df82f67753' (freq_y: 64, freq_x:␣
↪64)>
dask.array<mean_agg-aggregate, shape=(64, 64), dtype=float64, chunksize=(32, 32)>
Coordinates:
  * freq_y          (freq_y) float64 -0.5 -0.4844 -0.4688 -0.4531 -0.4375 ...
  * freq_x          (freq_x) float64 -0.5 -0.4844 -0.4688 -0.4531 -0.4375 ...
    freq_y_spacing  float64 0.01562
    freq_x_spacing  float64 0.01562
```

[19]:
```
fig, ax = plt.subplots()
ps.plot(ax=ax, norm=colors.LogNorm(), vmin=6.5e-4, vmax=7.5e-4)
```

[19]:
```
<matplotlib.collections.QuadMesh at 0x1210117b8>
```



[ ]:

1.8 Realistic example using outputs from MITgcm

This example requires the understanding of xgcm.grid and xmitgcm.open_mdsdataset.

[1]:
```
import numpy as np
import xarray as xr
import os.path as op
import xrft
from dask.diagnostics import ProgressBar
from xmitgcm import open_mdsdataset
from xgcm.grid import Grid
from matplotlib import colors, ticker
import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
ddir = '/swot/SUM05/takaya/MITgcm/channel/runs/'
```

One year of daily-averaged output from MITgcm.

[3]:
```
ys20, dy20 = (60,1)
dt = 8e2
df = 108
ts = int(360*86400*ys20/dt+df)
te = int(360*86400*(ys20+dy20)/dt+df)
ds = open_mdsdataset(op.join(ddir,'zerores_20km_MOMbgc'), grid_dir=op.join(ddir,'20km_
↪grid'),
                     iters=range(ts,te,df), prefix=['MOMtave'], delta_t=dt
                     ).sel(YC=slice(5e5,15e5), YG=slice(5e5,15e5))
ds
```

```
/home/takaya/xmitgcm/xmitgcm/utils.py:314: UserWarning: Not sure what to do with rlev = L
  warnings.warn("Not sure what to do with rlev = " + rlev)
/home/takaya/xmitgcm/xmitgcm/mds_store.py:235: FutureWarning: iteration over an xarray.
↪Dataset will change in xarray v0.11 to only include data variables, not coordinates.␣
↪Iterate over the Dataset.variables property instead to preserve existing behavior in a␣
↪forwards compatible manner.
  for vname in ds:
```

[3]:
```
<xarray.Dataset>
Dimensions:  (XC: 50, XG: 50, YC: 50, YG: 51, Z: 40, Zl: 40, Zp1: 41, Zu: 40, time: 360)
Coordinates:
  * XC       (XC) >f4 10000.0 30000.0 50000.0 70000.0 90000.0 110000.0 ...
  * YC       (YC) >f4 510000.0 530000.0 550000.0 570000.0 590000.0 610000.0 ...
  * XG       (XG) >f4 0.0 20000.0 40000.0 60000.0 80000.0 100000.0 120000.0 ...
  * YG       (YG) >f4 500000.0 520000.0 540000.0 560000.0 580000.0 600000.0 ...
  * Z        (Z) >f4 -5.0 -15.0 -25.0 -36.0 -49.0 -64.0 -81.5 -102.0 -126.0 ...
  * Zp1      (Zp1) >f4 0.0 -10.0 -20.0 -30.0 -42.0 -56.0 -72.0 -91.0 -113.0 ...
  * Zu       (Zu) >f4 -10.0 -20.0 -30.0 -42.0 -56.0 -72.0 -91.0 -113.0 ...
  * Zl       (Zl) >f4 0.0 -10.0 -20.0 -30.0 -42.0 -56.0 -72.0 -91.0 -113.0 ...
    rA       (YC, XC) >f4 dask.array<shape=(50, 50), chunksize=(50, 50)>
    dxG      (YG, XC) >f4 dask.array<shape=(51, 50), chunksize=(51, 50)>
    dyG      (YC, XG) >f4 dask.array<shape=(50, 50), chunksize=(50, 50)>
    Depth    (YC, XC) >f4 dask.array<shape=(50, 50), chunksize=(50, 50)>
    rAz      (YG, XG) >f4 dask.array<shape=(51, 50), chunksize=(51, 50)>
    dxC      (YC, XG) >f4 dask.array<shape=(50, 50), chunksize=(50, 50)>
    dyC      (YG, XC) >f4 dask.array<shape=(51, 50), chunksize=(51, 50)>
    rAw      (YC, XG) >f4 dask.array<shape=(50, 50), chunksize=(50, 50)>
    rAs      (YG, XC) >f4 dask.array<shape=(51, 50), chunksize=(51, 50)>
    drC      (Zp1) >f4 dask.array<shape=(41,), chunksize=(41,)>
    drF      (Z) >f4 dask.array<shape=(40,), chunksize=(40,)>
    PHrefC   (Z) >f4 dask.array<shape=(40,), chunksize=(40,)>
    PHrefF   (Zp1) >f4 dask.array<shape=(41,), chunksize=(41,)>
    hFacC    (Z, YC, XC) >f4 dask.array<shape=(40, 50, 50), chunksize=(40, 50, 50)>
    hFacW    (Z, YC, XG) >f4 dask.array<shape=(40, 50, 50), chunksize=(40, 50, 50)>
    hFacS    (Z, YG, XC) >f4 dask.array<shape=(40, 51, 50), chunksize=(40, 51, 50)>
    iter     (time) int64 dask.array<shape=(360,), chunksize=(1,)>
  * time     (time) float64 1.866e+09 1.866e+09 1.866e+09 1.867e+09 ...
Data variables:
    UVEL     (time, Z, YC, XG) float32 dask.array<shape=(360, 40, 50, 50), chunksize=(1,␣
↪40, 50, 50)>
    VVEL     (time, Z, YG, XC) float32 dask.array<shape=(360, 40, 51, 50), chunksize=(1,␣
↪40, 51, 50)>
    WVEL     (time, Zl, YC, XC) float32 dask.array<shape=(360, 40, 50, 50), chunksize=(1,
↪ 40, 50, 50)>
    PHIHYD   (time, Z, YC, XC) float32 dask.array<shape=(360, 40, 50, 50), chunksize=(1,␣
↪40, 50, 50)>
    THETA    (time, Z, YC, XC) float32 dask.array<shape=(360, 40, 50, 50), chunksize=(1,␣
↪40, 50, 50)>
```

[4]:
```
grid = Grid(ds, periodic=['X'])
```

[5]:
```
u = ds.UVEL     #zonal velocity
v = ds.VVEL     #meridional velocity
```

(continues on next page)

(continued from previous page)

```
w = ds.WVEL     #vertical velocity
phi = ds.PHIHYD #hydrostatic pressure
```

1.8.1 Discrete Fourier Transform

We chunk the data along the `time` and `Z` axes to allow parallelized computation and detrend and window the data before taking the DFT along the horizontal axes.

[6]:
```
b = grid.diff(phi,'Z',boundary='fill')/grid.diff(phi.Z,'Z',boundary='fill')
with ProgressBar():
    what = xrft.dft(w.chunk({'time':1,'Zl':1}),
                    dim=['XC','YC'], detrend='linear', window=True).compute()
    bhat = xrft.dft(b.chunk({'time':1,'Zl':1}),
                    dim=['XC','YC'], detrend='linear', window=True).compute()
bhat
```

```
/home/takaya/xrft/xrft/xrft.py:272: FutureWarning: xarray.DataArray.__contains__␣
↪currently checks membership in DataArray.coords, but in xarray v0.11 will change to␣
↪check membership in array values.
  elif d in da:
```

```
[########################################] | 100% Completed |  3min 18.9s
[########################################] | 100% Completed |  3min 20.1s
```

[6]:
```
<xarray.DataArray 'rechunk-merge-20d2920474ad47b75b05955c0456f69d' (time: 360, Zl: 40,␣
↪freq_YC: 50, freq_XC: 50)>
array([[[[ 2.801601e-03+8.771196e-17j, ..., -1.076925e-03-1.451031e-03j],
         ...,
         [-6.912831e-04-1.127047e-03j, ..., -1.114039e-03+8.825552e-04j]],

        ...,

        [[ 3.786092e-07-9.317362e-21j, ..., -7.612376e-07-3.355050e-07j],
         ...,
         [ 1.314610e-06+7.461259e-07j, ..., -1.471702e-06-5.475275e-07j]]],


       ...,


       [[[-3.941056e-04-1.888680e-16j, ..., -6.151166e-04+1.212955e-03j],
         ...,
         [-3.724654e-04+6.164418e-04j, ...,  1.445227e-03-2.389259e-04j]],

        ...,

        [[ 3.398755e-07-5.251604e-20j, ..., -7.561893e-07-1.006210e-06j],
         ...,
         [ 3.863488e-07+5.592156e-07j, ...,  1.252672e-06+1.153922e-06j]]]])
Coordinates:
  * time             (time) float64 1.866e+09 1.866e+09 1.866e+09 1.867e+09 ...
  * Zl               (Zl) >f4 0.0 -10.0 -20.0 -30.0 -42.0 -56.0 -72.0 -91.0 ...
```

(continues on next page)

(continued from previous page)

```
  * freq_YC          (freq_YC) float64 -2.5e-05 -2.4e-05 -2.3e-05 -2.2e-05 ...
  * freq_XC          (freq_XC) float64 -2.5e-05 -2.4e-05 -2.3e-05 -2.2e-05 ...
    freq_XC_spacing  float64 1e-06
    freq_YC_spacing  float64 1e-06
```

1.8.2 Power spectrum

We compute the surface eddy kinetic energy spectrum.

[8]:
```
with ProgressBar():
    uhat2 = xrft.power_spectrum(grid.interp(u,'X')[:,0].chunk({'time':1}),
                                dim=['XC','YC'], detrend='linear', window=True).compute()
    vhat2 = xrft.power_spectrum(grid.interp(v,'Y',boundary='fill')[:,0].chunk({'time':1}
↪),
                                dim=['XC','YC'], detrend='linear', window=True).compute()
ekehat = .5*(uhat2 + vhat2)
ekehat
```

```
/home/takaya/xrft/xrft/xrft.py:272: FutureWarning: xarray.DataArray.__contains__␣
↪currently checks membership in DataArray.coords, but in xarray v0.11 will change to␣
↪check membership in array values.
  elif d in da:
```

```
[########################################] | 100% Completed |  6.7s
[########################################] | 100% Completed |  6.4s
```

[8]:
```
<xarray.DataArray (time: 360, freq_YC: 50, freq_XC: 50)>
array([[[0.656013, 0.634131, ..., 0.373603, 0.634131],
        [0.420887, 0.593453, ..., 1.585473, 1.477422],
        ...,
        [1.743543, 0.468147, ..., 2.274391, 3.250841],
        [0.420887, 1.477422, ..., 1.285897, 0.593453]],

       [[0.005765, 0.126508, ..., 0.363493, 0.126508],
        [0.099985, 0.140124, ..., 0.49843 , 0.109594],
        ...,
        [1.436623, 0.598675, ..., 1.692357, 0.797681],
        [0.099985, 0.109594, ..., 0.497809, 0.140124]],

       ...,

       [[0.063022, 0.463507, ..., 0.839914, 0.463507],
        [0.161973, 0.310822, ..., 1.181991, 0.372522],
        ...,
        [0.122832, 0.415118, ..., 0.231018, 0.244315],
        [0.161973, 0.372522, ..., 0.500013, 0.310822]],

       [[0.140999, 0.475876, ..., 1.034042, 0.475876],
        [0.032224, 0.080152, ..., 1.088543, 0.660645],
        ...,
        [0.545666, 0.291697, ..., 4.745674, 1.533154],
        [0.032224, 0.660645, ..., 0.817556, 0.080152]]])
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * time             (time) float64 1.866e+09 1.866e+09 1.866e+09 1.867e+09 ...
  * freq_YC          (freq_YC) float64 -2.5e-05 -2.4e-05 -2.3e-05 -2.2e-05 ...
  * freq_XC          (freq_XC) float64 -2.5e-05 -2.4e-05 -2.3e-05 -2.2e-05 ...
    freq_XC_spacing  float64 1e-06
    freq_YC_spacing  float64 1e-06
```

1.8.3 Isotropic wavenumber spectrum

We now isotropize the spectrum:

[11]:
```
with ProgressBar():
    uiso2 = xrft.isotropic_powerspectrum(grid.interp(u,'X')[0,0],
                                         dim=['XC','YC'], detrend='linear', window=True).
↪compute()
    viso2 = xrft.isotropic_powerspectrum(grid.interp(v,'Y',boundary='fill')[0,0],
                                         dim=['XC','YC'], detrend='linear', window=True).
↪compute()
ekeiso = .5*(uiso2 + viso2)
ekeiso
```

```
[########################################] | 100% Completed |  0.1s
```

```
/home/takaya/xrft/xrft/xrft.py:428: RuntimeWarning: invalid value encountered in true_
↪divide
  kr = np.bincount(kidx, weights=K.ravel()) / area
/home/takaya/xrft/xrft/xrft.py:433: RuntimeWarning: invalid value encountered in true_
↪divide
  / area) * kr
```

```
[########################################] | 100% Completed |  0.1s
```

[11]:
```
<xarray.DataArray (freq_r: 13)>
array([         nan, 6.735224e+01, 1.488857e+02, 4.130706e+01, 1.541406e+01,
       9.217845e+00, 5.425922e+00, 1.887154e+00, 5.665645e-01, 2.813448e-01,
       6.721589e-02, 2.577505e-02, 7.507335e-03])
Coordinates:
  * freq_r   (freq_r) float64 nan 1.358e-06 3.36e-06 5.571e-06 7.73e-06 ...
```

We plot u, v, $\hat{u}^2+$

[22]:
```
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(20,4))
fig.set_tight_layout(True)
u[0,0].plot(ax=axes[0])
v[0,0].plot(ax=axes[1])
im = axes[2].pcolormesh(ekehat.freq_XC*1e3, ekehat.freq_YC*1e3, ekehat[0],
                        norm=colors.LogNorm())
axes[3].plot(ekeiso.freq_r*1e3, ekeiso)
cbar = fig.colorbar(im, ax=axes[2])
cbar.set_label(r'[m$^2$ s$^{-2}$]')
axes[3].set_xscale('log')
axes[3].set_yscale('log')
axes[2].set_xlabel(r'k [cpkm]')
```

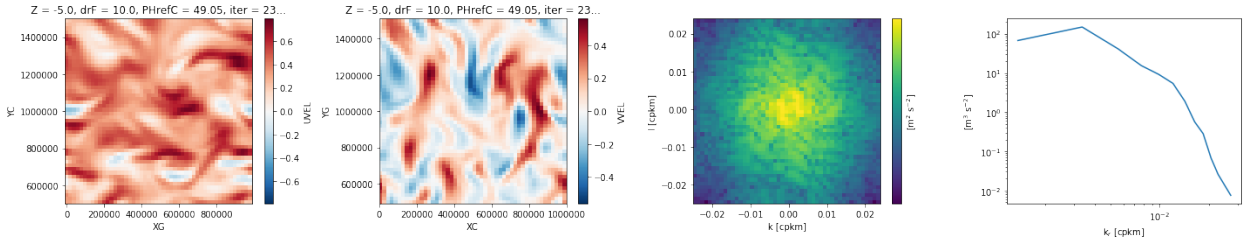
(continues on next page)

(continued from previous page)

```
axes[2].set_ylabel(r'l [cpkm]')
axes[3].set_xlabel(r'k$_r$ [cpkm]')
axes[3].set_ylabel(r'[m$^3$ s$^{-2}$]')
```

[22]:
```
Text(0,0.5,'[m$^3$ s$^{-2}$]')
```

```
/home/takaya/miniconda3/envs/uptodate/lib/python3.6/site-packages/matplotlib/scale.py:
↪111: RuntimeWarning: invalid value encountered in less_equal
  out[a <= 0] = -1000
/home/takaya/miniconda3/envs/uptodate/lib/python3.6/site-packages/matplotlib/figure.py:
↪2022: UserWarning: This figure includes Axes that are not compatible with tight_layout,
↪ so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



1.8.4 Cross Spectrum

We calculate the cross correlation between vertical velocity (w) and buoyancy (b):

[31]:
```
with ProgressBar():
    whatbhat = xrft.cross_spectrum(w.chunk({'time':1,'Zl':1}), b.chunk({'time':1,'Zl':1}
↪),
                                   dim=['XC','YC'], detrend='linear', window=True,␣
↪density=False).compute()
whatbhat
```

```
/home/takaya/xrft/xrft/xrft.py:272: FutureWarning: xarray.DataArray.__contains__␣
↪currently checks membership in DataArray.coords, but in xarray v0.11 will change to␣
↪check membership in array values.
  elif d in da:
```

```
[########################################] | 100% Completed |  7min 48.4s
```

[31]:
```
<xarray.DataArray (time: 360, Zl: 40, freq_YC: 50, freq_XC: 50)>
array([[[[ 6.217574e-11, ...,  5.227157e-11],
         ...,
         [-1.960930e-12, ...,  1.145311e-11]],

        ...,

        [[ 2.433719e-11, ...,  4.670022e-11],
         ...,
         [-9.319683e-11, ..., -7.301667e-11]]],


       ...,
```

(continues on next page)

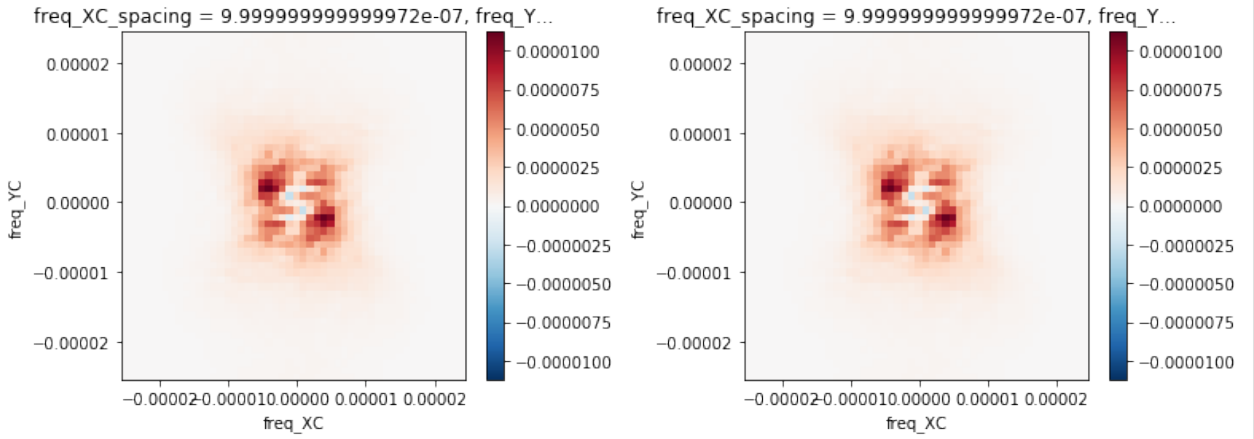
(continued from previous page)

```
       [[[-8.180808e-12, ...,  1.913746e-11],
         ...,
         [ 4.396894e-12, ..., -1.844566e-12]],

        ...,

        [[-1.291420e-11, ...,  2.346000e-11],
         ...,
         [ 2.565280e-11, ...,  4.489265e-11]]]])
Coordinates:
  * time             (time) float64 1.866e+09 1.866e+09 1.866e+09 1.867e+09 ...
  * Zl               (Zl) >f4 0.0 -10.0 -20.0 -30.0 -42.0 -56.0 -72.0 -91.0 ...
  * freq_YC          (freq_YC) float64 -2.5e-05 -2.4e-05 -2.3e-05 -2.2e-05 ...
  * freq_XC          (freq_XC) float64 -2.5e-05 -2.4e-05 -2.3e-05 -2.2e-05 ...
    freq_XC_spacing  float64 1e-06
    freq_YC_spacing  float64 1e-06
```

[32]:
```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(11,4))
fig.set_tight_layout(True)
(what*np.conjugate(bhat)).real[:,:8].mean(['time','Zl']).plot(ax=ax1)
whatbhat[:,:8].mean(['time','Zl']).plot(ax=ax2)
```

[32]:
```
<matplotlib.collections.QuadMesh at 0x7f10409c7ba8>
```

```
/home/takaya/miniconda3/envs/uptodate/lib/python3.6/site-packages/matplotlib/figure.py:
↪2022: UserWarning: This figure includes Axes that are not compatible with tight_layout,
↪ so results might be incorrect.
  warnings.warn("This figure includes Axes that are not compatible "
```



We see that $\hat{w}\hat{b}^*$ and `xrft.cross_spectrum`(w, b) are equivalent.

[ ]:

1.9 What's New

1.9.1 v0.3.0 (18 February 2021)

Enhancements

- Implemented the inverse discrete Fourier transform `idft`. By Frederic Nouguier
- Allowed windowing other than the Hann function. By Takaya Uchida
- Allowed parallelization of isotropizing the spectrum via `numpy_groupies`. By Takaya Uchida
- Implemented proper amplitude correction for real Fourier transform and windowed data. By Dougie Squire

1.9.2 v0.2.0 (10 April 2019)

Enhancements

- Allowed `dft` and `power_spectrum` functions to support real Fourier transforms. (**:issue:`57`**) By Takaya Uchida and Tom Nicholas.
- Implemented `cross_phase` function to calculate the phase difference between two signals as a function of frequency. By Tom Nicholas.
- Allowed `isotropic_powerspectrum` function to support arrays with up to four dimensions. (**:issue:`9`**) By Takaya Uchida

Warning: Python 2.7 is no longer supported in `xrft`. For the more details, see:

- Python 3 Statement
- Tips on porting to Python 3

1.10 API reference

This page provides an auto-generated summary of xrft's API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

Note: None of xrft's functions will work correctly in the presence of NaNs or missing data. It's the user's responsibility to ensure data are free of NaN or that NaNs have been filled somehow.

1.10.1 xrft

xrft.xrft.**cross_phase**(*da1*, *da2*, *dim=None*, *true_phase=True*, ***kwargs*)

Calculates the cross-phase between da1 and da2.

Returned values are in [-pi, pi].

$$da1' = da1 - \overline{da1};\;\; da2' = da2 - \overline{da2}$$

$$cp = extArg[\mathbb{F}(da1')^*, \mathbb{F}(da2')]$$

Parameters

da1 [*xarray.DataArray*] The data to be transformed

da2 [*xarray.DataArray*] The data to be transformed

dim [str or sequence of str, optional] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed.

true_phase [boolean] If True, the phase information is retained. Set explicitly true_phase = False in cross_spectrum arguments list to ensure future compatibility with numpy-like behavior where the coordinates are disregarded.

kwargs [dict][see xrft.fft for argument list]

xrft.xrft.**cross_spectrum**(*da1*, *da2*, *dim=None*, *real_dim=None*, *scaling='density'*, *window_correction=False*, *true_phase=True*, ***kwargs*)

Calculates the cross spectra of da1 and da2.

$$da1' = da1 - \overline{da1};\;\; da2' = da2 - \overline{da2}$$

$$cs = \mathbb{F}(da1')\mathbb{F}(da2')^*$$

Parameters

da1 [*xarray.DataArray*] The data to be transformed

da2 [*xarray.DataArray*] The data to be transformed

dim [str or sequence of str, optional] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed.

real_dim [str, optional] Real Fourier transform will be taken along this dimension.

scaling [str, optional] If 'density', it will normalize the output to power spectral density If 'spectrum', it will normalize the output to power spectrum

window_correction [boolean] If True, it will correct for the energy reduction resulting from applying a non-uniform window. This is the default behaviour of many tools for computing power spectrum (e.g scipy.signal.welch and scipy.signal.periodogram). If scaling = 'spectrum', correct the amplitude of peaks in the spectrum. This ensures, for example, that the peak in the one-sided power spectrum of a 10 Hz sine wave with RMS**2 = 10 has a magnitude of 10. If scaling = 'density', correct for the energy (integral) of the spectrum. This ensures, for example, that the power spectral density integrates to the square of the RMS of the signal (ie that Parseval's theorem is satisfied). Note that in most cases, Parseval's theorem will only be approximately satisfied with this correction as it assumes that the signal being windowed is independent of the window. The correction becomes more accurate as the width of the window gets large in comparison with any noticeable period in the signal. If False, the spectrum gives a representation of the power in the windowed signal. Note that when True, Parseval's theorem may only be approximately satisfied.

true_phase [boolean] If True, the phase information is retained. Set explicitly true_phase = False in cross_spectrum arguments list to ensure future compatibility with numpy-like behavior where the coordinates are disregarded.

kwargs [dict][see xrft.fft for argument list]

xrft.xrft.**dft**(*da*, *dim=None*, *true_phase=False*, *true_amplitude=False*, ***kwargs*)
Deprecated function. See fft doc

xrft.xrft.**fft**(*da*, *spacing_tol=0.001*, *dim=None*, *real_dim=None*, *shift=True*, *detrend=None*, *window=None*, *true_phase=True*, *true_amplitude=True*, *chunks_to_segments=False*, *prefix='freq_'*, ***kwargs*)
Perform discrete Fourier transform of xarray data-array *da* along the specified dimensions.

$$daft = \mathbb{F}(da - \overline{da})$$

Parameters

da [*xarray.DataArray*] The data to be transformed

spacing_tol: float, optional Spacing tolerance. Fourier transform should not be applied to uneven grid but this restriction can be relaxed with this setting. Use caution.

dim [str or sequence of str, optional] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed. If the inputs are dask arrays, the arrays must not be chunked along these dimensions.

real_dim [str, optional] Real Fourier transform will be taken along this dimension.

shift [bool, default] Whether to shift the fft output. Default is *True*, unless *real_dim is not None*, in which case shift will be set to False always.

detrend [{None, 'constant', 'linear'}] If *constant*, the mean across the transform dimensions will be subtracted before calculating the Fourier transform (FT). If *linear*, the linear least-square fit will be subtracted before the FT. For *linear*, only dims of length 1 and 2 are supported.

window [str, optional] Whether to apply a window to the data before the Fourier transform is taken. A window will be applied to all the dimensions in dim. Please follow *scipy.signal.windows*' naming convention.

true_phase [bool, optional] If set to False, standard fft algorithm is applied on signal without consideration of coordinates. If set to True, coordinates location are correctly taken into account to evaluate Fourier Tranforrm phase and fftshift is applied on input signal prior to fft (fft algorithm intrinsically considers that input signal is on fftshifted grid).

true_amplitude [bool, optional] If set to True, output is multiplied by the spacing of the transformed variables to match theoretical FT amplitude. If set to False, amplitude regularisation by spacing is not applied (as in numpy.fft)

chunks_to_segments [bool, optional] Whether the data is chunked along the axis to take FFT.

prefix [str] The prefix for the new transformed dimensions.

Returns

daft [*xarray.DataArray*] The output of the Fourier transformation, with appropriate dimensions.

xrft.xrft.**fit_loglog**(*x*, *y*)
Fit a line to isotropic spectra in log-log space

Parameters

x [*numpy.array*] Coordinate of the data

y [*numpy.array*] data

Returns

y_fit [*numpy.array*] The linear fit

a [float64] Slope of the fit

b [float64] Intercept of the fit

xrft.xrft.**idft**(*daft*, *dim=None*, *true_phase=False*, *true_amplitude=False*, ***kwargs*)
Deprecated function. See ifft doc

xrft.xrft.**ifft**(*daft*, *spacing_tol=0.001*, *dim=None*, *real_dim=None*, *shift=True*, *true_phase=True*, *true_amplitude=True*, *chunks_to_segments=False*, *prefix='freq_'*, *lag=None*, ***kwargs*)
Perform inverse discrete Fourier transform of xarray data-array *daft* along the specified dimensions.

$$da = \mathbb{F}(daft - \overline{daft})$$

Parameters

daft [*xarray.DataArray*] The data to be transformed

spacing_tol: float, optional Spacing tolerance. Fourier transform should not be applied to uneven grid but this restriction can be relaxed with this setting. Use caution.

dim [str or sequence of str, optional] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed.

real_dim [str, optional] Real Fourier transform will be taken along this dimension.

shift [bool, default] Whether to shift the fft output. Default is *True*.

chunks_to_segments [bool, optional] Whether the data is chunked along the axis to take FFT.

prefix [str] The prefix for the new transformed dimensions.

true_phase [bool, optional] If set to False, standard ifft algorithm is applied on signal without consideration of coordinates order. If set to True, coordinates are correctly taken into account to evaluate Inverse Fourier Tranforrm phase and fftshift is applied on input signal prior to ifft (ifft algorithm intrinsically considers that input signal is on fftshifted grid).

true_amplitude [bool, optional] If set to True, output is divided by the spacing of the transformed variables to match theoretical IFT amplitude. If set to False, amplitude regularisation by spacing is not applied (as in numpy.ifft)

lag [None, float or sequence of float and/or None, optional] Output coordinates of transformed dimensions will be shifted by corresponding lag values and correct signal phasing will be preserved if true_phase is set to True. If lag is None (default), 'direct_lag' attributes of each dimension is used (or set to zero if not found). If defined, lag must have same length as dim. If lag is a sequence, a None element means that 'direct_lag' attribute will be used for the corresponding dimension Manually set lag to zero to get output coordinates centered on zero.

Returns

da [*xarray.DataArray*] The output of the Inverse Fourier transformation, with appropriate dimensions.

xrft.xrft.**isotropic_cross_spectrum**(*da1*, *da2*, *spacing_tol=0.001*, *dim=None*, *shift=True*, *detrend=None*, *scaling='density'*, *window=None*, *window_correction=False*, *nfactor=4*, *truncate=False*, ***kwargs*)
Calculates the isotropic spectrum from the two-dimensional power spectrum by taking the azimuthal average.

$$extiso_{cs} = k_r N^{-1} \sum_N (\mathbb{F}(da1')\mathbb{F}(da2')^*)$$

where N is the number of azimuthal bins.

Note: the method is not lazy does trigger computations.

Parameters

da1 [*xarray.DataArray*] The data to be transformed

da2 [*xarray.DataArray*] The data to be transformed

spacing_tol: float (default) Spacing tolerance. Fourier transform should not be applied to uneven grid but this restriction can be relaxed with this setting. Use caution.

dim [list (optional)] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed.

shift [bool (optional)] Whether to shift the fft output.

detrend [str (optional)] If *constant*, the mean across the transform dimensions will be subtracted before calculating the Fourier transform (FT). If *linear*, the linear least-square fit will be subtracted before the FT.

density [list (optional)] If true, it will normalize the spectrum to spectral density

window [str (optional)] Whether to apply a window to the data before the Fourier transform is taken. Please adhere to scipy.signal.windows for naming convention.

nfactor [int (optional)] Ratio of number of bins to take the azimuthal averaging with the data size. Default is 4.

truncate [bool, optional] If True, the spectrum will be truncated for wavenumbers larger than the Nyquist wavenumber.

Returns

iso_cs [*xarray.DataArray*] Isotropic cross spectrum

xrft.xrft.**isotropic_power_spectrum**(*da*, *spacing_tol=0.001*, *dim=None*, *shift=True*, *detrend=None*, *scaling='density'*, *window=None*, *window_correction=False*, *nfactor=4*, *truncate=False*, ***kwargs*)

Calculates the isotropic spectrum from the two-dimensional power spectrum by taking the azimuthal average.

$$extiso_{ps} = k_r N^{-1} \sum_N |\mathbb{F}(da')|^2$$

where N is the number of azimuthal bins.

Note: the method is not lazy does trigger computations.

Parameters

da [*xarray.DataArray*] The data to be transformed

spacing_tol: float, optional Spacing tolerance. Fourier transform should not be applied to uneven grid but this restriction can be relaxed with this setting. Use caution.

dim [list, optional] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed.

shift [bool, optional] Whether to shift the fft output.

detrend [str, optional] If *constant*, the mean across the transform dimensions will be subtracted before calculating the Fourier transform (FT). If *linear*, the linear least-square fit will be subtracted before the FT.

density [list, optional] If true, it will normalize the spectrum to spectral density

window [str, optional] Whether to apply a window to the data before the Fourier transform is taken. Please adhere to scipy.signal.windows for naming convention.

nfactor [int, optional] Ratio of number of bins to take the azimuthal averaging with the data size. Default is 4.

truncate [bool, optional] If True, the spectrum will be truncated for wavenumbers larger than the Nyquist wavenumber.

Returns

iso_ps [*xarray.DataArray*] Isotropic power spectrum

xrft.xrft.**isotropize**(*ps*, *fftdim*, *nfactor=4*, *truncate=True*, *complx=False*)

Isotropize a 2D power spectrum or cross spectrum by taking an azimuthal average.

$$extiso_{ps} = k_r N^{-1} \sum_N |\mathbb{F}(da')|^2$$

where N is the number of azimuthal bins.

Parameters

ps [*xarray.DataArray*] The power spectrum or cross spectrum to be isotropized.

fftdim [list] The fft dimensions overwhich the isotropization must be performed.

nfactor [int, optional] Ratio of number of bins to take the azimuthal averaging with the data size. Default is 4.

truncate [bool, optional] If True, the spectrum will be truncated for wavenumbers larger than the Nyquist wavenumber.

complx [bool, optional] If True, isotropize allows for complex numbers.

xrft.xrft.**power_spectrum**(*da*, *dim=None*, *real_dim=None*, *scaling='density'*, *window_correction=False*, ***kwargs*)

Calculates the power spectrum of da.

da' = da - overline{da} .. math:: ps = mathbb{F}(da') {mathbb{F}(da')}^*

Parameters

da [*xarray.DataArray*] The data to be transformed

dim [str or sequence of str, optional] The dimensions along which to take the transformation. If *None*, all dimensions will be transformed.

real_dim [str, optional] Real Fourier transform will be taken along this dimension.

scaling [str, optional] If 'density', it will normalize the output to power spectral density If 'spectrum', it will normalize the output to power spectrum

window_correction [boolean] If True, it will correct for the energy reduction resulting from applying a non-uniform window. This is the default behaviour of many tools for computing power spectrum (e.g scipy.signal.welch and scipy.signal.periodogram). If scaling = 'spectrum', correct the amplitude of peaks in the spectrum. This ensures, for example, that the peak in the one-sided power spectrum of a 10 Hz sine wave with RMS**2 = 10 has a magnitude of 10. If scaling = 'density', correct for the energy (integral) of the spectrum. This ensures, for example, that the power spectral density integrates to the square of the RMS of the signal (ie that Parseval's theorem is satisfied). Note that in most cases, Parseval's theorem will only be approximately satisfied with this correction as it assumes that the signal being windowed is independent of the window. The correction becomes more accurate as the width of the window gets large in comparison with any noticeable period in the signal.

If False, the spectrum gives a representation of the power in the windowed signal. Note that when True, Parseval's theorem may only be approximately satisfied.

kwargs [dict][see xrft.fft for argument list]

1.10.2 detrend

You also may wish to use xrft's detrend function on its own.

Functions for detrending xarray data.

`xrft.detrend.detrend`(*da*, *dim*, *detrend_type='constant'*)

Detrend a DataArray

Parameters

da [xarray.DataArray] The data to detrend

dim [str or list] Dimensions along which to apply detrend. Can be either one dimension or a list with two dimensions. Higher-dimensional detrending is not supported. If dask data are passed, the data must be chunked along dim.

detrend_type [{'constant', 'linear'}] If `constant`, a constant offset will be removed from each dim. If `linear`, a linear least-squares fit will be estimated and removed from the data.

Returns

da [xarray.DataArray] The detrended data.

Notes

This function will act lazily in the presence of dask arrays on the input.

1.10.3 padding

Pad and unpad arrays and its coordinates so they can be used for computing FFTs.

Functions to pad and unpad a N-dimensional regular grid

`xrft.padding.pad`(*da*, *pad_width=None*, *mode='constant'*, *stat_length=None*, *constant_values=0*, *end_values=None*, *reflect_type=None*, ***pad_width_kwargs*)

Pad array with evenly spaced coordinates

Wraps the `xarray.DataArray.pad()` method but also pads the evenly spaced coordinates by extrapolation using the same coordinate spacing. The `pad_width` used for each coordinate is stored as one of its attributes.

Parameters

da [`xarray.DataArray`] Array to be padded. The coordinates along which the array will be padded must be evenly spaced.

pad_width [mapping of hashable to tuple of int] Mapping with the form of {dim: (pad_before, pad_after)} describing the number of values padded along each dimension. {dim: pad} is a shortcut for pad_before = pad_after = pad

mode [str, default: "constant"] One of the following string values (taken from numpy docs). - constant: Pads with a constant value. - edge: Pads with the edge values of array. - linear_ramp: Pads with the linear ramp between end_value and the

array edge value.

- maximum: Pads with the maximum value of all or part of the vector along each axis.
- mean: Pads with the mean value of all or part of the vector along each axis.
- median: Pads with the median value of all or part of the vector along each axis.
- minimum: Pads with the minimum value of all or part of the vector along each axis.
- reflect: Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.
- symmetric: Pads with the reflection of the vector mirrored along the edge of the array.
- wrap: Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

stat_length [int, tuple or mapping of hashable to tuple, default: None] Used in 'maximum', 'mean', 'median', and 'minimum'. Number of values at edge of each axis used to calculate the statistic value. {dim_1: (before_1, after_1), … dim_N: (before_N, after_N)} unique statistic lengths along each dimension. ((before, after),) yields same before and after statistic lengths for each dimension. (stat_length,) or int is a shortcut for before = after = statistic length for all axes. Default is `None`, to use the entire axis.

constant_values [scalar, tuple or mapping of hashable to tuple, default: 0] Used in 'constant'. The values to set the padded values for each axis. `{dim_1: (before_1, after_1), ... dim_N: (before_N, after_N)}` unique pad constants along each dimension. `((before, after),)` yields same before and after constants for each dimension. `(constant,)` or `constant` is a shortcut for `before = after = constant` for all dimensions. Default is 0.

end_values [scalar, tuple or mapping of hashable to tuple, default: 0] Used in 'linear_ramp'. The values used for the ending value of the linear_ramp and that will form the edge of the padded array. `{dim_1: (before_1, after_1), ... dim_N: (before_N, after_N)}` unique end values along each dimension. `((before, after),)` yields same before and after end values for each axis. `(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes. Default is 0.

reflect_type [{"even", "odd"}, optional] Used in "reflect", and "symmetric". The "even" style is the default with an unaltered reflection around the edge value. For the "odd" style, the extended part of the array is created by subtracting the reflected values from two times the edge value.

****pad_width_kwargs** The keyword arguments form of `pad_width`. One of `pad_width` or `pad_width_kwargs` must be provided.

Returns

da_padded [`xarray.DataArray`]

Examples

```
>>> import xarray as xr
>>> da = xr.DataArray(
...     [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
...     coords={"x": [0, 1, 2], "y": [-5, -4, -3]},
...     dims=("y", "x"),
... )
>>> da_padded = pad(da, x=2, y=1)
```

(continues on next page)

(continued from previous page)

```
>>> da_padded
<xarray.DataArray (y: 5, x: 7)>
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 2, 3, 0, 0],
       [0, 0, 4, 5, 6, 0, 0],
       [0, 0, 7, 8, 9, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
Coordinates:
  * x        (x) int64 -2 -1 0 1 2 3 4
  * y        (y) int64 -6 -5 -4 -3 -2
>>> da_padded.x
<xarray.DataArray 'x' (x: 7)>
array([-2, -1,  0,  1,  2,  3,  4])
Coordinates:
  * x        (x) int64 -2 -1 0 1 2 3 4
Attributes:
    pad_width:  2
>>> da_padded.y
<xarray.DataArray 'y' (y: 5)>
array([-6, -5, -4, -3, -2])
Coordinates:
  * y        (y) int64 -6 -5 -4 -3 -2
Attributes:
    pad_width:  1
```

Asymmetric padding

```
>>> da_padded = pad(da, x=(1, 4))
>>> da_padded
<xarray.DataArray (y: 3, x: 8)>
array([[0, 1, 2, 3, 0, 0, 0, 0],
       [0, 4, 5, 6, 0, 0, 0, 0],
       [0, 7, 8, 9, 0, 0, 0, 0]])
Coordinates:
  * x        (x) int64 -1 0 1 2 3 4 5 6
  * y        (y) int64 -5 -4 -3
>>> da_padded.x
<xarray.DataArray 'x' (x: 8)>
array([-1,  0,  1,  2,  3,  4,  5,  6])
Coordinates:
  * x        (x) int64 -1 0 1 2 3 4 5 6
Attributes:
    pad_width:  (1, 4)
```

xrft.padding.**unpad**(*da*, *pad_width=None*, ***pad_width_kwargs*)

Unpad an array and its coordinates

Undo the padding process of the `xrft.pad()` function by slicing the passed `xarray.DataArray` and its coordinates.

Parameters

da [`xarray.DataArray`] Padded array. The coordinates along which the array will be padded must be evenly spaced.

Returns

da_unpaded [xarray.DataArray] Unpadded array.

pad_width [mapping of hashable to tuple of int (optional)] Mapping with the form of {dim: (pad_before, pad_after)} describing the number of values padded along each dimension. {dim: pad} is a shortcut for pad_before = pad_after = pad. If None, then the *pad_width* for each coordinate is read from their pad_width attribute.

****pad_width_kwargs (optional)** The keyword arguments form of pad_width. Pass pad_width or pad_width_kwargs.

See also:

xrft.pad()

Examples

```
>>> import xarray as xr
>>> da = xr.DataArray(
...     [[1, 2, 3], [4, 5, 6], [7, 8, 9]],
...     coords={"x": [0, 1, 2], "y": [-5, -4, -3]},
...     dims=("y", "x"),
... )
>>> da_padded = pad(da, x=2, y=1)
>>> da_padded
<xarray.DataArray (y: 5, x: 7)>
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 2, 3, 0, 0],
       [0, 0, 4, 5, 6, 0, 0],
       [0, 0, 7, 8, 9, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
Coordinates:
  * x        (x) int64 -2 -1 0 1 2 3 4
  * y        (y) int64 -6 -5 -4 -3 -2
>>> unpad(da_padded)
<xarray.DataArray (y: 3, x: 3)>
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Coordinates:
  * x        (x) int64 0 1 2
  * y        (y) int64 -5 -4 -3
```

Custom pad_width

```
>>> unpad(da_padded, x=1, y=1)
<xarray.DataArray (y: 3, x: 5)>
array([[0, 1, 2, 3, 0],
       [0, 4, 5, 6, 0],
       [0, 7, 8, 9, 0]])
Coordinates:
  * x        (x) int64 -1 0 1 2 3
  * y        (y) int64 -5 -4 -3
```

PYTHON MODULE INDEX

X

INDEX