## Overview

### 1 Contents
- 3

### Bibliography
- 37

### Python Module Index
- 39
xESMF is a Python package for regridding. It is

- **Powerful**: It uses ESMF/ESMPy as backend and can regrid between general curvilinear grids with all ESMF regridding algorithms, such as bilinear, conservative and nearest neighbour.

- **Easy-to-use**: It abstracts away ESMF’s complicated infrastructure and provides a simple, high-level API, compatible with xarray as well as basic numpy arrays.

- **Fast**: It is faster than ESMPy’s original Fortran regridding engine (surprise! here’s benchmark), due to the use of the highly-optimized Scipy sparse matrix library.

---

**Note**: xESMF is at early stage of development and will keep improving in the future. The major regridding 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.
1. Why inventing a new regridding package

1.1 For scientific correctness

Traditional interpolation routines, such as interp2d in Scipy and interp2 in MATLAB, assume flat 2D planes and do not consider the spherical geometry of the earth. They are great for image processing, but will produce incorrect/distorted results for geospatial data.

Also, traditional interpolation algorithms are typically based on piecewise polynomials ("splines"). While being highly accurate in terms of error convergence, they often lack desired physical properties such as conservation (total mass should be conserved) and monotonicity (air density cannot go negative).

1.2 For emerging new grid types

Non-orthogonal grids are becoming popular in numerical models (Staniforth and Thuburn 2012), but traditional tools often assume standard lat-lon grids.

xESMF can regrid between general curvilinear (i.e. quadrilateral or “logically rectilinear”) grids, like

- The Cubed-Sphere grid in GFDL-FV3
- The Latitude-Longitude-Cap grid in MITgcm
- The Lambert Conformal grid in WRF

However, xESMF does not yet support non-quadrilateral grids, like the hexagonal grid in MPAS. See Irregular meshes for more information.

1.3 For usability and simplicity

Current geospatial regridding tools tend to have non-trivial learning curves. xESMF tries to be simple and intuitive. Instead of inventing a new data structure, it relies on well-established standards (numpy and xarray), so users don’t
need to learn a bunch of new syntaxes or even a new software stack.

xESMF can track the metadata in `xarray.DataArray (example)`, but it can also work with the basic `numpy.ndarray (example)`. This means any Python users can use it easily, even if being unfamiliar with `xarray`.

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

### 1.2 Other geospatial regridding tools

Here is a brief overview of other regridding tools that the author is aware of (for geospatial data on the sphere, excluding traditional image resizing functions). They are all great tools and have helped the author a lot in both scientific research and xESMF development. Check them out if xESMF cannot suit your needs.

- **ESMF (Fortran package)**

  Although its name “Earth System Modeling Framework” doesn’t indicate a regridding functionality, it actually contains a very powerful regridding engine. It is widely used in Earth System Models (ESMs), serving as both the software infrastructure and the regridder for transforming data between the atmosphere, ocean, and land components. It can deal with general irregular meshes, in either 2D or 3D.

  ESMF is a huge beast, containing one million lines of source code. Even just compiling it requires some effort. It is more for building ESMs than for data analysis.

- **ESMPy (Python interface to ESMF)**

  ESMPy provides a much simpler way to use ESMF’s regridding functionality. The greatest thing is, it is pre-compiled as a conda package, so you can install it with one-click and don’t have to go through the daunting compiling process on your own.

  However, ESMPy is a complicated Python API that controls a huge Fortran beast hidden underneath. It is not as intuitive as native Python packages, and even a simple regridding task requires more than 10 lines of arcane code.

  That’s why I made xESMF. If you want to involve in xESMF development you need to know ESMPy. Check out this nice tutorial before going to the official doc.

- **TempestRemap (C++ package)**

  A pretty modern and powerful package, supporting arbitrary-order conservative remapping. It can also generate cubed-sphere grids on the fly and can be modified to support many cubed-sphere grid variations (example, only if you can read C++).

- **SCRIP (Fortran package)**

  An old package, once popular but no longer maintained (long live SCRIP). You should not use it now, but should know that it exists. Newer regridding packages often follow its standards – you will see “SCRIP format” here and there, for example in ESMF or TempestRemap.

- **Regridder in NCL (NCAR Command Language)**

  Has bilinear and conservative algorithms for rectilinear grids, and also supports some specialized curvilinear grids. There is also an ESMF wrapper that works for more grid types.

- **Regridder in NCO (command line tool)**

- **Regridder in Iris (Python package)**

- **Regridder in UV-CDAT (Python package)**
1.3 Current limitations

1.3.1 Irregular meshes

xESMF only supports quadrilateral grids and doesn’t support weirder grids like triangular or hexagonal meshes.

ESMPy is actually able to deal with general irregular meshes (example), but designing an elegant front-end for that is very challenging. Plain 2D arrays cannot describe irregular meshes. There needs to be additional information for connectivity, as suggested by UGRID Conventions.

xarray’s data model, although powerful, can only describe quadrilateral grids (including multi-tile quadrilateral grids like the cubed-sphere). If there is an elegant data model in Python for irregular meshes, interfacing that with ESMPy should not be super difficult.

1.3.2 Parallel regridding

xESMF currently only runs in serial. Parallel options are being investigated. See issue #3.

1.3.3 Vector regridding

Like almost all regridding packages, xESMF assumes scalar fields. The most common way to remap winds is to rotate/re-decompose the wind components (U and V) to the new direction, and then regrid each component individually using a scalar regridding function.

Exact conservation of vector properties (like divergence and vorticity) is beyond the scope of almost all regridding packages. Using bilinear algorithm on each component should lead to OK results in most cases.

1.4 Installation

1.4.1 The quickest way

xESMF requires Python>=3.5. The major dependencies are xarray and ESMPy. The best way to install them is using Anaconda:

```bash
$ conda install xarray
$ conda install -c conda-forge esmpy
```

Then install xesmf:

```bash
$ pip install xesmf
```

Done! You can go to the next tutorial.

1.4.2 Notes for Windows users

The ESMPy conda package is currently only available for Linux and Mac OSX. Windows users can try the Linux subsystem or docker-miniconda.

Installing scientific software on Windows can often be a pain, and Docker is a pretty good workaround. It takes some time to learn but worths the effort. Check out this tutorial on using Docker with Anaconda.

This problem is being investigated. See this issue.
1.4.3 Install xESMF from GitHub repo

To get the latest version that is not uploaded to PyPI yet:

```
$ pip install --upgrade git+https://github.com/JiaweiZhuang/xESMF.git
```

Developers can track source code change:

```
$ git clone https://github.com/JiaweiZhuang/xESMF.git
$ cd xESMF
$ pip install -e .
```

1.5 Regrid between rectilinear grids

1.5.1 Prepare data

Input data

We regrid xarray’s built-in demo data. This data is also used by xarray plotting tutorial.

```
[2]: ds = xr.tutorial.open_dataset('air_temperature')  # use xr.tutorial.load_dataset() for
     →xarray<v0.11.0
```

```
[2]: <xarray.Dataset>
    Dimensions:  (lat: 25, lon: 53, time: 2920)
    Coordinates:
    * lat  (lat) float32 75.0 72.5 70.0 67.5 65.0 62.5 60.0 57.5 55.0 52.5 ...  
    * lon  (lon) float32 200.0 202.5 205.0 207.5 210.0 212.5 215.0 217.5 ...  
    * time (time) datetime64[ns] 2013-01-01T00:02:06.757437440 ...  
    Data variables:
    air    (time, lat, lon) float64 241.2 242.5 243.5 244.0 244.1 243.9 ...  
    Attributes:
    Conventions:  COARDS
    title:  4x daily NMC reanalysis (1948)
    description:  Data is from NMC initialized reanalysis\n(4x/day). These a...
    platform:  Model
    references:  http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanaly...
```

It is the air temperature data over US with 2920 time frames. Let’s plot the first frame:

```
[3]: dr = ds['air']  # get a DataArray
```

```
[4]: ax = plt.axes(projection=ccrs.PlateCarree())
    dr.isel(time=0).plot.pcolormesh(ax=ax, vmin=230, vmax=300);
    ax.coastlines();
```
Input grid

Its grid resolution is $2.5^\circ \times 2.5^\circ$:

```python
[5]: ds['lat'].values, ds['lon'].values
```

```python
[5]: (array([ 75. , 72.5, 70. , 67.5, 65. , 62.5, 60. , 57.5, 55. ,
            52.5, 50. , 47.5, 45. , 42.5, 40. , 37.5, 35. , 32.5,
            30. , 27.5, 25. , 22.5, 20. , 17.5, 15. ], dtype=float32),
   array([ 200. , 202.5, 205. , 207.5, 210. , 212.5, 215. , 217.5,
          220. , 222.5, 225. , 227.5, 230. , 232.5, 235. , 237.5,
          240. , 242.5, 245. , 247.5, 250. , 252.5, 255. , 257.5,
          260. , 262.5, 265. , 267.5, 270. , 272.5, 275. , 277.5,
          280. , 282.5, 285. , 287.5, 290. , 292.5, 295. , 297.5,
          300. , 302.5, 305. , 307.5, 310. , 312.5, 315. , 317.5,
          320. , 322.5, 325. , 327.5, 330. ], dtype=float32))
```

Output grid

Say we want to downsample it to $0.25^\circ \times 0.5^\circ$. Just define the output grid as an `xarray` `Dataset`:

```python
[6]: ds_out = xr.Dataset({'lat': ('lat', np.arange(15, 75.01, 0.25)),
                        'lon': ('lon', np.arange(200, 330.01, 0.5))},
                        )
```

```python
[6]: <xarray.Dataset>
Dimensions: (lat: 241, lon: 261)
Coordinates:
  * lat (lat) float64 15.0 15.25 15.5 15.75 16.0 16.25 16.5 16.75 17.0 ...
  * lon (lon) float64 200.0 200.5 201.0 201.5 202.0 202.5 203.0 203.5 ...
Data variables:
  *empty*
```

1.5. Regrid between rectilinear grids
**1.5.2 Perform regridding**

Make a regridder by `xe.Regridder(grid_in, grid_out, method)`. `grid` is just an xarray `DataSet` containing `lat` and `lon` values. In most cases, 'bilinear' should be good enough. For other methods see [Comparison of 5 regridding algorithms](#).

```python
[7]: regridder = xe.Regridder(ds, ds_out, 'bilinear')
regridder # print basic regridder information.
```

Create weight file: bilinear_25x53_241x261.nc

```python
[7]: xESMF Regridder
Regridding algorithm: bilinear
Weight filename: bilinear_25x53_241x261.nc
Reuse pre-computed weights? False
Input grid shape: (25, 53)
Output grid shape: (241, 261)
Output grid dimension name: ('lat', 'lon')
Periodic in longitude? False
```

The regridder says it can transform data from shape (25, 53) to shape (241, 261).

Regrid the `DataArray` is straightforward:

```python
[8]: dr_out = regridder(dr)
dr_out
```

```python
[xarray.DataArray 'air' (time: 2920, lat: 241, lon: 261)>
array([[ 296.29 , 0. , ..., 0. , 296.6 ],
       [ 296.250991, 296.347386, ..., 296.509328, 296.459968],
       ...,  
       [ 241.600595, 241.702929, ..., 238.032375, 238.670016],
       [ 241.2 , 241.462205, ..., 237.980361, 238.6 ]],

       ...,  

       [ 296.29 , 0. , ..., 0. , 296.6 ],
       [ 296.280994, 296.454975, ..., 296.423187, 296.449966],
       ...,  
       [ 242.250034, 242.369363, ..., 235.329657, 235.789998],
       [ 242.1 , 242.22131 , ..., 235.359671, 235.8 ]],

       ...,  

       [ 297.79 , 0. , ..., 0. , 295.19 ],
       [ 297.639963, 297.767262, ..., 295.178632, 295.109982],
       ...,  
       [ 246.200094, 246.011777, ..., 244.559336, 244.739989],
       [ 245.79 , 245.936198, ..., 244.629364, 244.79 ]],

       ...,  

       [ 297.69 , 0. , ..., 0. , 295.69 ],
       [ 297.529963, 297.619325, ..., 295.740447, 295.639989],
       ...,  
       [ 245.570112, 245.418376, ..., 241.697697, 241.779998],
       [ 245.09 , 244.934201, ..., 241.729711, 241.79 ]])
```

The horizontal shape is now (241, 261), as expected. The regridding operation broadcasts over extra dimensions.
(time here), so there are still 2920 time frames. lon and lat coordinate values are updated accordingly, and the value of the extra dimension time is kept the same as input.

**Important note:** Extra dimensions must be on the left, i.e. (time, lev, lat, lon) is correct but (lat, lon, time, lev) would not work. Most data sets should have (lat, lon) on the right (being the fastest changing dimension in the memory). If not, use `DataArray.transpose` or `numpy.transpose` to preprocess the data.

### 1.5.3 Check results on 2D map

The regridding result is consistent with the original data, with a much finer resolution:

```python
[9]: ax = plt.axes(projection=ccrs.PlateCarree())
dr_out.isel(time=0).plot.pcolormesh(ax=ax, vmin=230, vmax=300);
ax.coastlines();
```

![2D map](image)

### 1.5.4 Check broadcasting over extra dimensions

xESMF tracks coordinate values over extra dimensions, since horizontal regridding should not affect them.

```python
[10]: dr_out['time']
[10]: <xarray.DataArray 'time' (time: 2920)>
Coordinates:
  * time (time) datetime64[ns] 2013-01-01T00:02:06.757437440 ...
Attributes:
  standard_name: time
  long_name: Time
```

```python
[11]: # exactly the same as input
xr.testing.assert_identical(dr_out['time'], ds['time'])
```

We can plot the time series at a specific location, to make sure the broadcasting is correct:
1.5.5 Clean-up

xESMF saves the regridder to the current directory so you don’t need to re-compute it next time (see *Save time by reusing regridder*). If you don’t need it anymore, you can just delete it:

```python
regridder.clean_weight_file()  # regridder.c + TAB would bring-up the command
```

Remove file bilinear_25x53_241x261.nc

1.6 Regrid between curvilinear grids

1.6.1 Prepare data

Input data

Here we regrid the built-in “rasm” demo data. This data is used by another *xarray* tutorial.
It is the surface air temperature data, with \texttt{nan} over the ocean.

It is the surface air temperature data, with \texttt{nan} over the ocean.

1.6. Regrid between curvilinear grids
Input grid

xc and yc are longitude and latitude values. They are both 2D arrays, describing a curvilinear grid over high-latitudes. Note that it is totally fine for a grid to span over the south or north pole. ESMF performs regridding in the Cartesian space (x, y, z) so there will be no polar singularities.
We need to rename the coordinate names to `lon` and `lat` because xESMF has no way to guess variable meaning.

```python
[6]: ds = ds.rename({'xc': 'lon', 'yc': 'lat'})
```

```python
[6]: <xarray.Dataset>
  Dimensions: (time: 36, x: 275, y: 205)
  Coordinates:
    * time (time) datetime64[ns] 1980-09-16T12:00:00 1980-10-17 ...
    lon (y, x) float64 189.2 189.4 189.6 189.7 189.9 190.1 190.2 190.4 ...
    lat (y, x) float64 16.53 16.78 17.02 17.27 17.51 17.76 18.0 18.25 ...
  Dimensions without coordinates: x, y
  Data variables:
    Tair (time, y, x) float64 nan nan nan nan nan nan nan nan nan nan ...
  Attributes:
    title: /workspace/jhamman/processed/R1002RBRxaaa01a/1...
    institution: U.W.
    source: RACM R1002RBRxaaa01a
    output_frequency: daily
    output_mode: averaged
    convention: CF-1.4
    references: Based on the initial model of Liang et al., 19...
    comment: Output from the Variable Infiltration Capacity...
    nco_openmp_thread_number: 1
    NCO: "4.6.0"
    history: Tue Dec 27 14:15:22 2016: ncatted -a dimension...
```

### Output grid

Say we want to regrid it to a global $4^\circ \times 5^\circ$ grid. xESMF provides a shortcut to make this output grid.

```python
[7]: ds_out = xe.util.grid_global(5, 4)
```

```python
[7]: <xarray.Dataset>
  Dimensions: (x: 72, x_b: 73, y: 45, y_b: 46)
  Coordinates:
    lon (y, x) float64 -177.5 -172.5 -167.5 -162.5 -157.5 -152.5 -147.5 ...
    lat (y, x) float64 -88.0 -88.0 -88.0 -88.0 -88.0 -88.0 -88.0 -88.0 ...
    lon_b (y_b, x_b) int64 -180 -175 -170 -165 -160 -155 -150 -145 ...
    lat_b (y_b, x_b) int64 -90 -90 -90 -90 -90 -90 -90 -90 ...
  Dimensions without coordinates: x, x_b, y, y_b
  Data variables:
    *empty*
```

The output coordinates are all 2D arrays. They happen to be a rectilinear grid in this case (`lat` is constant over `x` axis, and `lon` is constant over `y` axis), but you can use 2D arrays to specify any curvilinear grids.

### 1.6.2 Perform regridding

Regridding is straightforward, just like the previous example.

```python
[8]: regridder = xe.Regridder(ds, ds_out, 'bilinear')
dr_out = regridder(dr)
```

Overwrite existing file: bilinear_205x275_45x72.nc

You can set `reuse_weights=True` to save computing time.

---

1.6. Regrid between curvilinear grids
1.6.3 Check results

Extra dimensions and coordinate values are all correct, like in the previous example.

```python
[9]: dr_out
[9]: <xarray.DataArray 'Tair' (time: 36, y: 45, x: 72)>
array([[ 0., 0., ..., 0., 0.],
       [ 0., 0., ..., 0., 0.],
       ...
       [ nan, nan, ..., nan, nan],
       [ nan, nan, ..., nan, nan]],

       ...
       [[ 0., 0., ..., 0., 0.],
       [ 0., 0., ..., 0., 0.],
       ...
       [ nan, nan, ..., nan, nan],
       [ nan, nan, ..., nan, nan]],

       ...
       [[ 0., 0., ..., 0., 0.],
       [ 0., 0., ..., 0., 0.],
       ...
       [ nan, nan, ..., nan, nan],
       [ nan, nan, ..., nan, nan]],

       ...
       [[ 0., 0., ..., 0., 0.],
       [ 0., 0., ..., 0., 0.],
       ...
       [ nan, nan, ..., nan, nan],
       [ nan, nan, ..., nan, nan]],

       ...
       [[ 0., 0., ..., 0., 0.],
       [ 0., 0., ..., 0., 0.],
       ...
       [ nan, nan, ..., nan, nan],
       [ nan, nan, ..., nan, nan]],

       ...
       [[ 0., 0., ..., 0., 0.],
       [ 0., 0., ..., 0., 0.],
       ...
       [ nan, nan, ..., nan, nan],
       [ nan, nan, ..., nan, nan]],

       *
       time (time) datetime64[ns] 1980-09-16T12:00:00 1980-10-17 ...

       Coordinates:
       lon (y, x) float64 -177.5 -172.5 -167.5 -162.5 -157.5 -152.5 -147.5 ...
       lat (y, x) float64 -88.0 -88.0 -88.0 -88.0 -88.0 -88.0 -88.0 ...
       *
       time (time) datetime64[ns] 1980-09-16T12:00:00 1980-10-17 ...

       Dimensions without coordinates: y, x

       Attributes:
       regrid_method: bilinear
```

The regridding result is consistent with the original data, but now on a rectilinear grid with a coarser resolution. `nan` is mapped to `nan`.

```python
[10]: plt.figure(figsize=(12,4));
ax = plt.axes(projection=ccrs.PlateCarree())
dr_out[0].plot.pcolormesh(ax=ax, x='lon', y='lat');
ax.coastlines();
```
1.7 Use pure numpy array

Despite the “x” in its name (indicating xarray-compatible), xESMF can also work with basic numpy arrays. You don’t have to use xarray data structure if you don’t need to track metadata. As long as you have numpy arrays describing the input data and input/output coordinate values, you can perform regridding.

Code in this section is adapted from an xarray example.

```
[1]: # not importing xarray here!
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import xesmf as xe
```

1.7.1 Rectilinear grid

Input data

Just make some fake data.

```
[2]: data = np.arange(20).reshape(4, 5)
plt.pcolormesh(data)
[2]: <matplotlib.collections.QuadMesh at 0x1180e7b38>
```
Define grids

In previous examples we use xarray Dataset as input/output grids. But you can also use a simple dictionary:

```python
[3]: grid_in = {'lon': np.linspace(0, 40, 5),
             'lat': np.linspace(0, 20, 4)}

# output grid has a larger coverage and finer resolution
grid_out = {'lon': np.linspace(-20, 60, 51),
             'lat': np.linspace(-10, 30, 41)}
```

Perform regridding

```python
[4]: regridder = xe.Regridder(grid_in, grid_out, 'bilinear')
regridder.clean_weight_file()
regridder
Create weight file: bilinear_4x5_41x51.nc
Remove file bilinear_4x5_41x51.nc
```

The regridder here has no difference from the ones made from xarray Dataset. You can use it to regrid DataArray or just a basic numpy.ndarray:

```python
[5]: data_out = regridder(data)  # regrid a basic numpy array
data_out.shape
```
1.7. Use pure numpy array
For the output grid, just use a simple rectilinear one:

```python
[9]: lon_out_b = np.linspace(-30, 40, 36)  # bounds
lon_out = 0.5*(lon_out_b[1:]+lon_out_b[:-1])  # centers
lat_out_b = np.linspace(-20, 50, 36)
lat_out = 0.5*(lat_out_b[1:]+lat_out_b[:-1])
```

To use conservative algorithm, both input and output grids should contain 4 variables: lon, lat, lon_b, lon_b.

```python
[10]: grid_in = {'lon': lon, 'lat': lat,
             'lon_b': lon_b, 'lat_b': lat_b}
grid_out = {'lon': lon_out, 'lat': lat_out,
            'lon_b': lon_out_b, 'lat_b': lat_out_b}
```

### Regridding

```python
[11]: regridder = xe.Regridder(grid_in, grid_out, 'conservative')
regridder.clean_weight_file()
regridder
Create weight file: conservative_4x5_35x35.nc
Remove file conservative_4x5_35x35.nc
```

```python
[12]: data_out = regridder(data)
data_out.shape
```
1.7.3 All possible combinations

All $2 \times 2 \times 2 = 8$ combinations would work:

- **Input grid**: `xarray.Dataset` or `dict`
- **Output grid**: `xarray.Dataset` or `dict`
- **Input data**: `xarray.DataArray` or `numpy.ndarray`

The output data type will be the same as input data.

1.8 Comparison of 5 regridding algorithms

Those algorithms are available:

```python
[1]: method_list = ['bilinear', 'conservative', 'nearest_s2d', 'nearest_d2s', 'patch']
```

- Bilinear and conservative should be the most commonly used methods. They are both monotonic (i.e. will not create new maximum/minimum).
- Nearest neighbour methods, either source to destination (s2d) or destination to source (d2s), could be useful in special cases. Keep in mind that d2s is highly non-monotonic.
- Patch is ESMF’s unique method, producing highly smooth results but quite slow.

Detailed explanations are available on ESMPy documentation.
1.8.1 Preparation

```python
}[2]: %matplotlib inline
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import numpy as np
import xarray as xr
import xesmf as xe

[3]: ds_in = xe.util.grid_global(20, 15) # input grid
    ds_fine = xe.util.grid_global(4, 4) # high-resolution target grid
    ds_coarse = xe.util.grid_global(30, 20) # low-resolution target grid

Make a wave field that is widely used in regridding benchmarks.

[4]: ds_in['data'] = xe.data.wave_smooth(ds_in['lon'], ds_in['lat'])

```
```
<
```
<
```
```

[5]: ds_in['data'].plot()

```python

5]: def regrid(ds_in, ds_out, dr_in, method):
    """Convenience function for one-time regridding""
    regridder = xe.Regridder(ds_in, ds_out, method, periodic=True)
    dr_out = regridder(dr_in)
    regridder.clean_weight_file()
    return dr_out

```
When dealing with global grids, we need to set `periodic=True`, otherwise data along the meridian line will be missing.

### 1.8.2 Increasing resolution

```python
for method in method_list:
    %time ds_fine[method] = regrid(ds_in, ds_fine, ds_in['data'], method)
    print('')
```

Create weight file: bilinear_12x18_45x90_peri.nc
Remove file bilinear_12x18_45x90_peri.nc
CPU times: user 228 ms, sys: 14.1 ms, total: 242 ms
Wall time: 243 ms

Create weight file: conservative_12x18_45x90.nc
Remove file conservative_12x18_45x90.nc
CPU times: user 160 ms, sys: 5.67 ms, total: 165 ms
Wall time: 176 ms

Create weight file: nearest_s2d_12x18_45x90_peri.nc
Remove file nearest_s2d_12x18_45x90_peri.nc
CPU times: user 53.7 ms, sys: 2.92 ms, total: 56.6 ms
Wall time: 58.8 ms

Create weight file: nearest_d2s_12x18_45x90_peri.nc
Remove file nearest_d2s_12x18_45x90_peri.nc
CPU times: user 23.1 ms, sys: 1.89 ms, total: 25 ms
Wall time: 25.7 ms

Create weight file: patch_12x18_45x90_peri.nc
Remove file patch_12x18_45x90_peri.nc
CPU times: user 838 ms, sys: 39.3 ms, total: 878 ms
Wall time: 893 ms

Nearest neighbour algorithms are very fast while the patch method is quite slow.

```python
fig, axes = plt.subplots(3, 2, figsize=[8, 8])
for i, method in enumerate(method_list):
    ax = axes.flatten()[i]
    ds_fine[method].plot.pcolormesh(ax=ax)
    ax.set_title(method, fontsize=15)
plt.tight_layout()
```

1.8. Comparison of 5 regridding algorithms
When regridding from low-resolution to high-resolution, bilinear and patch will produce smooth results, while conservative and nearest_s2d will preserve the original coarse grid structure (although the data is now defined on a finer grid.).

nearest_d2s is quite different from others: One source point can be mapped to only one destination point. Because we have far less source points (on a low-resolution grid) than destination points (on a high-resolution grid), most destination points cannot receive any data so they just have zero values. Only the destination points that are closest to source points can receive data.

1.8.3 Decreasing resolution

```python
for method in method_list:
    ds_coarse[method] = regrid(ds_in, ds_coarse, ds_in['data'], method)
```

Create weight file: bilinear_12x18_9x12_peri.nc
Remove file bilinear_12x18_9x12_peri.nc

(continues on next page)
Create weight file: conservative_12x18_9x12.nc
Remove file conservative_12x18_9x12.nc
Create weight file: nearest_s2d_12x18_9x12_peri.nc
Remove file nearest_s2d_12x18_9x12_peri.nc
Create weight file: nearest_d2s_12x18_9x12_peri.nc
Remove file nearest_d2s_12x18_9x12_peri.nc
Create weight file: patch_12x18_9x12_peri.nc
Remove file patch_12x18_9x12_peri.nc

[10]: fig, axes = plt.subplots(3, 2, figsize=[8, 8])

for i, method in enumerate(method_list):
    ax = axes.flatten()[i]
    ds_coarse[method].plot.pcolormesh(ax=ax)
    ax.set_title(method, fontsize=15)

plt.tight_layout()

1.8. Comparison of 5 regridding algorithms
When regridding from high-resolution to low-resolution, all methods except nearest_d2s produce similar results here. But that’s largely because the input data is smooth. For real-world data, it is generally recommended to use conservative for upscaling, because it takes average over small source grid boxes, while bilinear and nearest_s2d effectively throw away most of source grid boxes.

nearest_d2s is again different: Every source point has to be mapped to a destination point. Because we have far more source points (on a high-resolution grid) than destination points (on a low-resolution grid), a single destination point will receive data from multiple source points, which can accumulate to a large value (notice the colorbar range).

1.9 Save time by reusing regridder

There is an important reason why the regridding is broken into two steps (making the regridder and perform regridding). For high-resolution grids, making the regridder (i.e. “computing regridding weights”, explained later) is quite computationally expensive, but performing regridding on data (“applying regridding weights”) is still pretty fast.

1.9.1 Prepare data

The grids in previous examples were all quite small and the regridding was almost instantaneous. Let’s try a large-ish grid here.

```
[1]: %matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr
import xesmf as xe

[2]: ds_in = xe.util.grid_2d(-120, 120, 0.4, # longitude range and resolution
                           -60, 60, 0.3) # latitude range and resolution

ds_in

[2]: <xarray.Dataset>
    Dimensions: (x: 600, x_b: 601, y: 400, y_b: 401)
    Coordinates:
      lon  (y, x) float64 -119.8 -119.4 -119.0 -118.6 -118.2 -117.8 ...  
      lat  (y, x) float64  -59.85 -59.85 -59.85 -59.85 -59.85 -59.85 ...  
      lon_b (y_b, x_b) float64 -120.0 -119.6 -119.2 -118.8 -118.4 -118.0 ...  
      lat_b (y_b, x_b) float64  -60.0 -60.0 -60.0 -60.0 -60.0 -60.0 ...  
    Dimensions without coordinates: x, x_b, y, y_b
    Data variables:
      *empty*

[3]: ds_out = xe.util.grid_2d(-120, 120, 0.6,  
                           -60, 60, 0.4)

ds_out

[3]: <xarray.Dataset>
    Dimensions: (x: 400, x_b: 401, y: 300, y_b: 301)
    Coordinates:
      lon  (y, x) float64 -119.7 -119.1 -118.5 -117.9 -117.3 -116.7 ...  
      lat  (y, x) float64  -59.8 -59.8 -59.8 -59.8 -59.8 -59.8 ...  
      lon_b (y_b, x_b) float64 -120.0 -119.4 -118.8 -118.2 -117.6 -117.0 ...  
      lat_b (y_b, x_b) float64  -60.0 -60.0 -60.0 -60.0 -60.0 -60.0 ...  
    Dimensions without coordinates: x, x_b, y, y_b
    Data variables:
      *empty*
Also make a large-ish 4D data, with multiple time frames and vertical levels.

```python
[4]:
ds_in.coords['time'] = np.arange(1, 11)
ds_in.coords['lev'] = np.arange(1, 51)
ds_in['data2D'] = xe.data.wave_smooth(ds_in['lon'], ds_in['lat'])
ds_in['data4D'] = ds_in['time'] * ds_in['lev'] * ds_in['data2D']
ds_in
```

```
<xarray.Dataset>
Dimensions: (lev: 50, time: 10, x: 600, x_b: 601, y: 400, y_b: 401)
Coordinates:
    lon  (y, x) float64 -119.8 -119.4 -119.0 -118.6 -118.2 -117.8 -117.4 ...
    lat  (y, x) float64 -59.85 -59.85 -59.85 -59.85 -59.85 -59.85 -59.85 ...
    lon_b (y_b, x_b) float64 -120.0 -119.6 -119.2 -118.8 -118.4 -118.0 ...
    lat_b (y_b, x_b) float64 -60.0 -60.0 -60.0 -60.0 -60.0 -60.0 -60.0 ...
    * time  (time) int64 1 2 3 4 5 6 7 8 9 10
    * lev   (lev) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
Dimensions without coordinates: x, x_b, y, y_b
Data variables:
    data2D  (y, x) float64 1.872 1.869 1.866 1.863 1.86 1.857 1.855 1.852 ...
    data4D  (time, lev, y, x) float64 1.872 1.869 1.866 1.863 1.86 1.857 ...
```

It is almost 1GB!

```python
[5]:
ds_in['data4D'].nbytes / 1e9 # Byte -> GB
[5]: 0.96
```

The data itself is not too interesting... We only focus on performance here.

```python
[6]:
plt.figure(figsize=[12, 3])

plt.subplot(121)
ds_in['data4D'].isel(time=0, lev=0).plot()
plt.title('2D field')

plt.subplot(122)
ds_in['data4D'].mean(dim=['x','y']).plot()
plt.title('extra dimensions to test broadcasting')
```

```
1.9.2 Build Regridder

Making a bilinear regridder takes ~7s on my Mac! ('conservative' would take even longer. Try it yourself.)

1.9. Save time by reusing regridder
```
1.9.3 Apply regridding

However, applying the regridder to 1GB of data only takes ~0.5s

1.9.4 Why applying regridding is so fast?

Most regridding algorithms (including all 5 algorithms in ESMF) are linear, i.e. the output data field is linearly dependent on the input data field. Any linear transform can be viewed as a matrix-vector multiplication $y = Ax$, where $A$ is a matrix containing regridding weights, and $x, y$ are input and output data fields flatten to 1D.

Computing the weight matrix $A$ is expensive, but $A$ only depends on input and output grids, not on input data. That means we can use the same $A$ on different input fields $x$, as long as the grid structure is not changed.

An xESMF regridder has an attribute $A$, i.e. the weight matrix.

It is typically very sparse, because a single destination point will only receive contribution from a small number of source points.
1.9.5 Retrieve regridder

When you open the notebook next time, instead of spending another ~7s on recomputing the weights, you can simply set `reuse_weights=True` to read existing weights from disk.

The weight file is typically pretty small (due to sparsity), so reading it is almost instantaneous.

```
[12]: %bash
du -sh bilinear_400x600_300x400.nc
    7.3M  bilinear_400x600_300x400.nc
```

```
[13]: %time
    regridder2 = xe.Regridder(ds_in, ds_out, 'bilinear', reuse_weights=True)

    Reuse existing file: bilinear_400x600_300x400.nc
    CPU times: user 23.4 ms, sys: 12.9 ms, total: 36.3 ms
    Wall time: 36.2 ms
```

The second-step, applying those weights to data, is just a matrix multiplication $y = Ax$. With highly-optimized sparse matrix multiplication library, it is blazingly fast.

```
[14]: %time
    dr_out2 = regridder2(ds_in['data4D'])

    CPU times: user 460 ms, sys: 164 ms, total: 624 ms
    Wall time: 628 ms
```

The retrieved regridder gives the same result as the first regridder.

```
[15]: xr.testing.assert_identical(dr_out, dr_out2)  # they are equal
```

For even larger grids, you might spend several minutes computing the weights. But once they are computed, you don’t have to do it again.

```
[16]: # don't have to clean it if you want to use it next time
    regridder2.clean_weight_file()

    Remove file bilinear_400x600_300x400.nc
```
1.10 xESMF backend usage and benchmark

xESMF isn’t just a wrapper of ESMPy. It only uses ESMPy to generate regridding weights, but has its own Scipy-based method for applying weights (see more about regridding weights).

We switch to the Scipy method because its serial performance is much higher than ESMPy’s own engine and can also reuse weights (issue#2). ESMPy’s native method is available in the backend, mainly for benchmarking Scipy results in unit tests.

Here we show how to use xESMF backend and compare the performance of two methods. Note that the backend is still pretty easy to use compared to the original ESMPy – it just doesn’t have a fancy API and cannot deal with xarray metadata.

```python
[1]: import os
import numpy as np
import xesmf as xe

# backend functions
from xesmf.backend import (esmf_grid, esmf_regrid_build,
esmf_regrid_apply, esmf_regrid_finalize)
from xesmf.smm import read_weights, apply_weights
```

### 1.10.1 Prepare data

We use the same data as in the reusing regridder example, but convert xarray DataSet to pure numpy arrays to work with the backend.

```python
[2]: ds_in = xe.util.grid_2d(-120, 120, 0.4,  # longitude range and resolution
                           -60, 60, 0.3)  # latitude range and resolution
    ds_out = xe.util.grid_2d(-120, 120, 0.6,
                            -60, 60, 0.4)
    ds_in.coords['time'] = np.arange(1, 11)
    ds_in.coords['lev'] = np.arange(1, 51)
    ds_in['data2D'] = xe.data.wave_smooth(ds_in['lon'], ds_in['lat'])
    ds_in['data4D'] = ds_in['time'] * ds_in['lev'] * ds_in['data2D']

[3]: # backend only accepts pure numpy array
    lon_in = ds_in['lon'].values
    lat_in = ds_in['lat'].values
    lon_out = ds_out['lon'].values
    lat_out = ds_out['lat'].values
    data_in = ds_in['data4D'].values
    data_in.shape

[3]: (10, 50, 400, 600)
```

### 1.10.2 Make ESMF Grid objects

```python
[4]: grid_in = esmf_grid(lon_in.T, lat_in.T)
    grid_out = esmf_grid(lon_out.T, lat_out.T)
```

This is a native ESMPy Grid object:
[5]: `type(grid_in)`

We pass the transpose (`lon.T`) because ESMPy prefers Fortran-ordering to C-ordering (see this issue).

[6]: `lon_in.flags` # numpy arrays are mostly C-ordered

```
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

[7]: `lon_in.T.flags` # a memory view on its transpose would be Fortran-ordered

```
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

### 1.10.3 Compute weights

[8]: `filename = 'test_weights.nc' # weight filename
if os.path.exists(filename):
    os.remove(filename) # ESMPy will crash if the file exists`

Computing weights takes ~7s, as in the [reusing regridder example](#).

[9]: `%time
regrid = esmf_regrid_build(grid_in, grid_out, 'bilinear',
    extra_dims=[50, 10], # reversed to Fortran-ordering
    filename=filename)

CPU times: user 7.06 s, sys: 382 ms, total: 7.44 s
Wall time: 7.57 s`

It returns a native ESMPy Regrid object:

[10]: `type(regrid)`

```
ESMF.api.regrid.Regrid
```

It also writes weights to disk so we can then read them back for Scipy.

```bash
%%bash
ncdump -h test_weights.nc
```

```
netcdf test_weights {
  dimensions:
    n_s = 480000 ;
  variables:
    double S(n_s) ;
    int col(n_s) ;
```

(continues on next page)
int row(n_s) ;
}

1.10.4 Apply weights using ESMPy backend

It takes ~3s with ESMPy’s native method.

```python
[12]: %%time
data_out_esmpy = esmf_regrid_apply(regrid, data_in.T).T
cpu times: user 2.35 s, sys: 662 ms, total: 3.01 s
wall time: 3.09 s
```

The first `.T` converts C-ordering to F-ordering for ESMPy, and the second `.T` converts the result back to C-ordering. It just gets a memory view and thus incurs almost no overhead.

```python
[13]: data_out_esmpy.flags

C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

```python
[14]: data_out_esmpy.shape

# broadcasted over extra dimensions
```

```
[14]: (10, 50, 300, 400)
```

1.10.5 Apply weights using Scipy backend

Read weights back for Scipy. `read_weights` needs to know the shape of the sparse matrix, i.e. how many points in input and output grids.

```python
[15]: A = read_weights(filename, lon_in.size, lon_out.size)
A
```

```
<120000x240000 sparse matrix of type '<class 'numpy.float64'>'
  with 480000 stored elements in COOrdinate format>
```

`apply_weights` needs to know shape of the output grid.

```python
[16]: Nlat_out, Nlon_out = lon_out.shape
Nlat_out, Nlon_out
```

```
(300, 400)
```

```python
[17]: %%time
data_out_scipy = apply_weights(A, data_in, Nlat_out, Nlon_out)
cpu times: user 443 ms, sys: 165 ms, total: 609 ms
wall time: 620 ms
```

It is several times faster than ESMPy’s native method. The conclusion seems to be pretty robust across different platforms (feel free to verify on your own), so we choose Scipy as the default backend.
A likely explanation for this performance discrepancy is, the original ESMF is optimized for large processor counts (~1000 CPUs) at the expense of serial performance (ESMF team, personal communication).

```python
[18]: data_out_scipy.shape  # broadcasted over extra dimensions
[18]: (10, 50, 300, 400)

[19]: np.testing.assert_equal(data_out_scipy, data_out_esmcpy)  # exactly the same

[20]: os.remove(filename)  # clean-up
```

### 1.11 User API

#### 1.11.1 Regridder

```python
class xesmf.frontend.Regridder(ds_in, ds_out, method, periodic=False, filename=None, reuse_weights=False)

__init__(ds_in, ds_out, method, periodic=False, filename=None, reuse_weights=False)
Make xESMF regridder

Parameters

ds_in, ds_out [xarray DataSet, or dictionary] Contain input and output grid coordinates. Look for variables lon, lat, and optionally lon_b, lat_b for conservative method. Shape can be 1D (Nlon,) and (Nlat,) for rectilinear grids, or 2D (Ny, Nx) for general curvilinear grids. Shape of bounds should be (N+1,) or (Ny+1, Nx+1).

method [str] Regridding method. Options are
  - 'bilinear'
  - 'conservative', need grid corner information
  - 'patch'
  - 'nearest_s2d'
  - 'nearest_d2s'


filename [str, optional] Name for the weight file. The default naming scheme is:

```
{method}_(Ny_in)x{Nx_in}_(Ny_out)x{Nx_out}.nc
```

  e.g. bilinear_400x600_300x400.nc

reuse_weights [bool, optional] Whether to read existing weight file to save computing time. False by default (i.e. re-compute, not reuse).

Returns

regridder [xESMF regridder object]```
**clean_weight_file()**
Remove the offline weight file on disk.

To save the time on re-computing weights, you can just keep the file, and set “reuse_weights=True” when initializing the regrider next time.

**__call__(a)**
Shortcut for regrid_numpy() and regrid_dataarray().

**Parameters**
- **a** [xarray DataArray or numpy array]

**Returns**
- **xarray DataArray or numpy array** Regridding results. Type depends on input.

**regrid_numpy (indata)**
Regrid pure numpy array. Shape requirement is the same as regrid_dataarray()

**Parameters**
- **indata** [numpy array]

**Returns**
- **outdata** [numpy array]

**regrid_dataarray (dr_in)**
Regrid xarray DataArray, track metadata.

**Parameters**
- **dr_in** [xarray DataArray] The rightmost two dimensions must be the same as ds_in. Can have arbitrary additional dimensions.

  **Examples of valid shapes**
  - (Nlat, Nlon), if ds_in has shape (Nlat, Nlon)
  - (N2, N1, Ny, Nx), if ds_in has shape (Ny, Nx)

**Returns**
- **dr_out** [xarray DataArray] On the same horizontal grid as ds_out, with extra dims in dr_in.

  Assuming ds_out has the shape of (Ny_out, Nx_out), examples of returning shapes are
  - (Ny_out, Nx_out), if dr_in is 2D
  - (N2, N1, Ny_out, Nx_out), if dr_in has shape (N2, N1, Ny, Nx)

### 1.11.2 util

desk3f.util.grid_2d (lon0_b, lon1_b, d_lon, lat0_b, lat1_b, d_lat)
2D rectilinear grid centers and bounds

**Parameters**
- **lon0_b, lon1_b** [float] Longitude bounds
- **d_lon** [float] Longitude step size, i.e. grid resolution
- **lat0_b, lat1_b** [float] Latitude bounds
- **d_lat** [float] Latitude step size, i.e. grid resolution
Returns
ds [xarray DataSet with coordinate values]

xesmf.util.grid_global(d_lon, d_lat)
Global 2D rectilinear grid centers and bounds

Parameters
d_lon [float] Longitude step size, i.e. grid resolution
d_lat [float] Latitude step size, i.e. grid resolution

Returns
ds [xarray DataSet with coordinate values]

1.11.3 data

Standard test data for regridding benchmark.

xesmf.data.wave_smooth(lon, lat)
Spherical harmonic with low frequency.

Parameters
lon, lat [2D numpy array or xarray DataArray] Longitude/Latitude of cell centers

Returns
f [2D numpy array or xarray DataArray depending on input] 2D wave field

Notes
Equation from [1] [2]:
\[ Y_2^2 = 2 + \cos^2(\theta) \cos(2\phi) \]

References
[1], [2]

1.12 Internal API

1.12.1 frontend

Frontend for xESMF, exposed to users.

xesmf.frontend.ds_to_ESMFgrid(ds, need_bounds=False, periodic=None, append=None)
Convert xarray DataSet or dictionary to ESMF.Grid object.

Parameters
ds [xarray DataSet or dictionary] Contains variables lon, lat, and optionally lon_b, lat_b
if need_bounds=True.
Shape should be (Nlat, Nlon) or (Ny, Nx), as normal C or Python ordering. Will be
then tranposed to F-ordered.
need_bounds [bool, optional] Need cell boundary values?
periodic [bool, optional] Periodic in longitude?

Returns
grid [ESMF.Grid object]

1.12.2 backend

Backend for xESMF. This module wraps ESMPy’s complicated API and can create ESMF Grid and Regrid objects only using basic numpy arrays.

General idea:
1) Only use pure numpy array in this low-level backend. xarray should only be used in higher-level APIs which interface with this low-level backend.
2) Use simple, procedural programming here. Because ESMPy Classes are complicated enough, building new Classes will make debugging very difficult.
3) Add some basic error checking in this wrapper level. ESMPy is hard to debug because the program often dies in the Fortran level. So it would be helpful to catch some common mistakes in Python level.

xesmf.backend.warn_f_contiguous(a)
Give a warning if input array is not Fortran-ordered.

ESMPy expects Fortran-ordered array. Passing C-ordered array will slow down performance due to memory rearrangement.

Parameters
a [numpy array]

xesmf.backend.warn_lat_range(lat)
Give a warning if latitude is outside of [-90, 90]

Longitude, on the other hand, can be in any range, since the it the transform is done in (x, y, z) space.

Parameters
lat [numpy array]

xesmf.backend.esmf_grid(lon, lat, periodic=False)
Create an ESMF.Grid object, for constructing ESMF.Field and ESMF.Regrid

Parameters
lon, lat [2D numpy array] Longitude/Latitude of cell centers.

Recommend Fortran-ordering to match ESMPy internal.

Shape should be (Nlon, Nlat) for rectilinear grid, or (Nx, Ny) for general quadrilateral grid.


Returns
grid [ESMF.Grid object]

xesmf.backend.add_corner(grid, lon_b, lat_b)
Add corner information to ESMF.Grid for conservative regridding.

Not needed for other methods like bilinear or nearest neighbour.
Parameters

grid [ESMF.Grid object] Generated by esmf_grid(). Will be modified in-place.

lon_b, lat_b [2D numpy array] Longitude/Latitude of cell corner Recommend Fortran-ordering to match ESMPy internal. Shape should be (Nlon+1, Nlat+1), or (Nx+1, Ny+1)

xesmf.backend.esmf_regrid_build(sourcegrid, destgrid, method, filename=None, extra_dims=None)
Create an ESMF.Regrid object, containing regridding weights.

Parameters

sourcegrid, destgrid [ESMF.Grid object] Source and destination grids.

    Should create them by esmf_grid() (with optionally add_corner()), instead of ESMPy’s original API.

method [str] Regridding method. Options are

    • ‘bilinear’
    • ‘conservative’, need grid corner information
    • ‘patch’
    • ‘nearest_s2d’
    • ‘nearest_d2s’

filename [str, optional] Offline weight file. Require ESMPy 7.1.0.dev38 or newer. With the weights available, we can use Scipy’s sparse matrix multiplication to apply weights, which is faster and more Pythonic than ESMPy’s online regridding.

extra_dims [a list of integers, optional] Extra dimensions (e.g. time or levels) in the data field

    This does NOT affect offline weight file, only affects online regrid.

    Extra dimensions will be stacked to the fastest-changing dimensions, i.e. following Fortran-like instead of C-like conventions. For example, if extra_dims=[Nlev, Ntime], then the data field dimension will be [Nlon, Nlat, Nlev, Ntime]

Returns

grid [ESMF.Grid object]

xesmf.backend.esmf_regrid_apply(regrid, indata)
Apply existing regridding weights to the data field, using ESMPy’s built-in functionality.

xesMF use Scipy to apply weights instead of this. This is only for benchmarking Scipy’s result and performance.

Parameters

regrid [ESMF.Regrid object] Contains the mapping from the source grid to the destination grid.

    Users should create them by esmf_regrid_build(), instead of ESMPy’s original API.

indata [numpy array of shape (Nlon, Nlat, N1, N2, ...)] Extra dimensions (N1, N2, ...) are specified in esmf_regrid_build().

    Recommend Fortran-ordering to match ESMPy internal.

Returns

outdata [numpy array of shape (Nlon_out, Nlat_out, N1, N2, ...)]
xesmf.backend.esmf_regrid_finalize(regrid)

Free the underlying Fortran array to avoid memory leak.

After calling destroy() on regrid or its fields, we cannot use the regrid method anymore, but the input and output data still exist.

Parameters

regrid [ESMF.Regrid object]

1.12.3 smm

Sparse matrix multiplication (SMM) using scipy.sparse library.

xesmf.smm.read_weights(filename, N_in, N_out)

Read regridding weights into a scipy sparse COO matrix.

Parameters

filename [str] Offline weight file generated by ESMPy.

N_in, N_out [integers] (N_out, N_in) will be the shape of the returning sparse matrix. They are the total number of grid boxes in input and output grids:

\[
N_{\text{in}} = Nx_{\text{in}} \times Ny_{\text{in}} \\
N_{\text{out}} = Nx_{\text{out}} \times Ny_{\text{out}}
\]

We need them because the shape cannot always be inferred from the largest column and row indices, due to unmapped grid boxes.

Returns

A [scipy sparse COO matrix.]

xesmf.smm.apply_weights(A, indata, Ny_out, Nx_out)

Apply regridding weights to data.

Parameters

A [scipy sparse COO matrix]

indata [numpy array of shape (... , Nlat, Nlon) or (... , Ny, Nx).] Should be C-ordered. Will be then tranposed to F-ordered.

Ny_out, Nx_out [integers] Output data shape for unflatten operation. For rectilinear grid, it is just \( (Nlat_{\text{out}}, Nlon_{\text{out}}) \).

Returns

outdata [numpy array of shape (... , Ny_out, Nx_out).] Extra dimensions are the same as indata. If input data is C-ordered, output will also be C-ordered.

X
xesmf.backend, 34
xesmf.data, 33
xesmf.frontend, 33
xesmf.smm, 36
xesmf.util, 32
Symbols
__call__() (xesmf.frontend.Regridder method), 32
__init__() (xesmf.frontend.Regridder method), 31

A
add_corner() (in module xesmf.backend), 34
apply_weights() (in module xesmf.smm), 36

C
clean_weight_file() (xesmf.frontend.Regridder method), 31

D
ds_to_ESMFgrid() (in module xesmf.frontend), 33

E
esmf_grid() (in module xesmf.backend), 34
esmf_regrid_apply() (in module xesmf.backend), 35
esmf_regrid_build() (in module xesmf.backend), 35
esmf_regrid_finalize() (in module xesmf.backend), 35

G
ggrid_2d() (in module xesmf.util), 32
ggrid_global() (in module xesmf.util), 33

R
read_weights() (in module xesmf.smm), 36
regrid_dataarray() (xesmf.frontend.Regridder method), 32
regrid_numpy() (xesmf.frontend.Regridder method), 32
Regridder (class in xesmf.frontend), 31

W
warn_f_contiguous() (in module xesmf.backend), 34
warn_lat_range() (in module xesmf.backend), 34
wave_smooth() (in module xesmf.data), 33

X
xesmf.backend (module), 34
xesmf.data (module), 33
xesmf.frontend (module), 33
xesmf.smm (module), 36
xesmf.util (module), 32