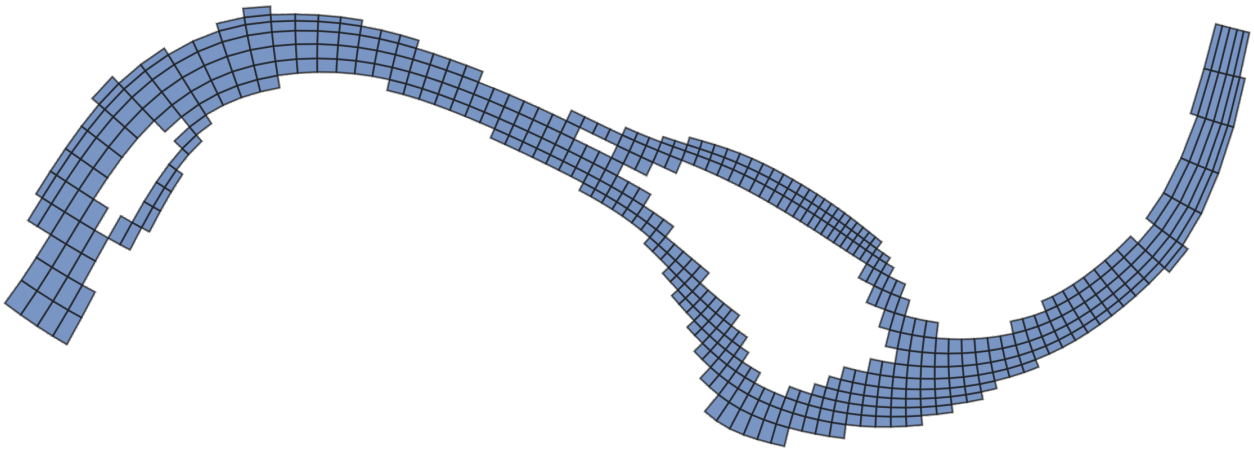
pygridtools Documentation

Release 0.3.0

Paul Hobson

Oct 17, 2018

Contents



PYGRIDTOOLS

A high-level interface for curvilinear-orthogonal grid generation, manipulation, and visualization.

Depends heavily on gridgen and pygridgen

The full documentation for this for library is here.

CHAPTER 1

Installation and Depedencies

conda-forge generously maintains Linux and Mac OS X conda builds of *pygridtools*.

Install with

```
conda install pygridtools --channel=conda-forge
```

Building (`gridgen-c`) on Windows has been a tough nut to crack and help is very much wanted in that department. Until we figure that out, you can do the following in the source directory.

```
conda create --name=grid python=3.6 numpy scipy pandas matplotlib shapely geopandas --
↪channel=conda-forge
pip install -e .
```

You won't be able to generate new grids, but you should be able to manipulate existing grids.

To create new grids on Linux or Mac OS, you'll need `pygridgen`

```
conda activate grid
conda install pygridgen --channel=conda-forge
```

If you want to use the interactive ipywidgets to manipulate grid parameters, you'll need a few elements of the jupyter ecosystem

```
conda activate grid
conda install notebook ipywidgets --channel=conda-forge
```

If you'd like to build the docs, you need a few more things

```
conda activate grid
conda install sphinx numpydoc sphinx_rtd_theme nbsphinx --channel=conda-forge
```

Finally, to fully run the tests, you need `pytest` and a few plugins

```
conda activate grid
conda install pytest pytest-mpl pytest-pep8 --channel=conda-forge
```

1.1 Grid Generation

If you wish to generate new grids from scratch, you'll need pygridgen, which is also available through the conda-forge channel.

```
conda install pygridgen --channel=conda-forge
```

The documentation *pygridgen* has a more detailed tutorial on generating new grids.

1.2 Testing

Tests are written using the *pytest* package. From the source tree, run them simply with by invoking `pytest` in a terminal. If you're editing the source code, it helps to have *pytest-pep8* installed to check code style.

Alternatively, from the source tree you can run `python check_pygridtools.py --strict` to run the units tests, style checker, and doctests.

1.3 Documentation

Building the HTML documentation requires:

- sphinx
- sphinx_rtd_theme
- numpydoc
- jupyter-notebook
- nbsphinx
- pandas
- seaborn

CHAPTER 2

Source code and Issue Tracker

The source code is available on Github at Geosyntec/pygridtools.

Please report bugs, issues, and ideas there.

CHAPTER 3

Contributing

1. Feedback is a huge contribution
2. Get in touch by creating an issue to make sure we don't duplicate work
3. Fork this repo
4. Submit a PR in a separate branch
5. Write a test (or two (or three))
6. Stick to PEP8-ish – I'm lenient on the 80 chars thing (<100 is probably a smart move though).

CHAPTER 4

Tutorials

4.1 Grid Generation Basics

This section will cover:

1. Loading and visualizing boundary data
2. Generating visualizing a basic grid
3. Adding focus to the grid

This is sorely lacking in detail and explanation. For more help on this matter, see the pygridgen documentation.

```
In [1]: %matplotlib inline
        import warnings
        warnings.simplefilter('ignore')

        import numpy as np
        import matplotlib.pyplot as plt
        import pandas
        import geopandas

        import pygridgen as pgg
        import pygridtools as pgt
```

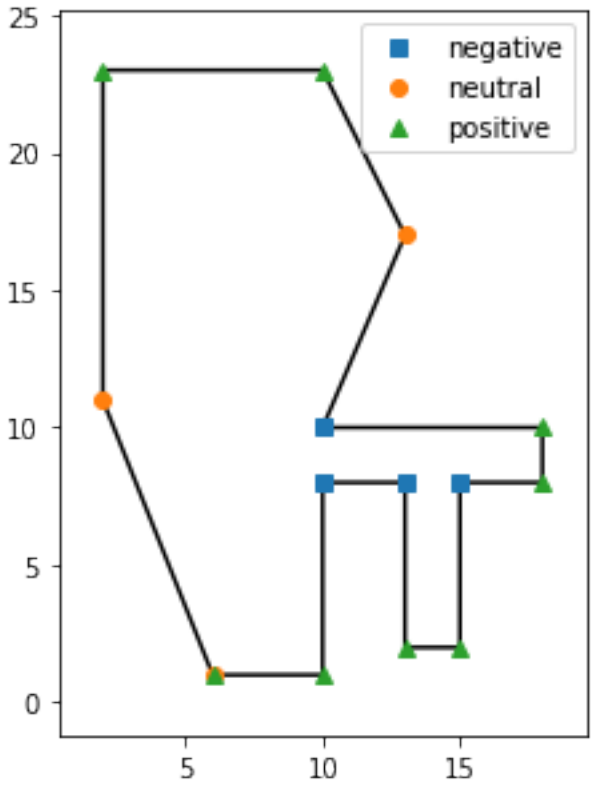
4.1.1 Loading and plotting the boundary data

```
In [2]: domain = geopandas.read_file('basic_data/domain.geojson')
        domain
```

```
Out[2]:     beta       geometry
        0      1    POINT (6 1)
        1      0   POINT (2 11)
        2      1   POINT (2 23)
        3      1  POINT (10 23)
        4      0  POINT (13 17)
```

```
5     -1  POINT (10 10)
6      1  POINT (18 10)
7      1   POINT (18 8)
8     -1   POINT (15 8)
9      1   POINT (15 2)
10     1   POINT (13 2)
11    -1   POINT (13 8)
12    -1   POINT (10 8)
13     1   POINT (10 1)
14     0    POINT (6 1)
```

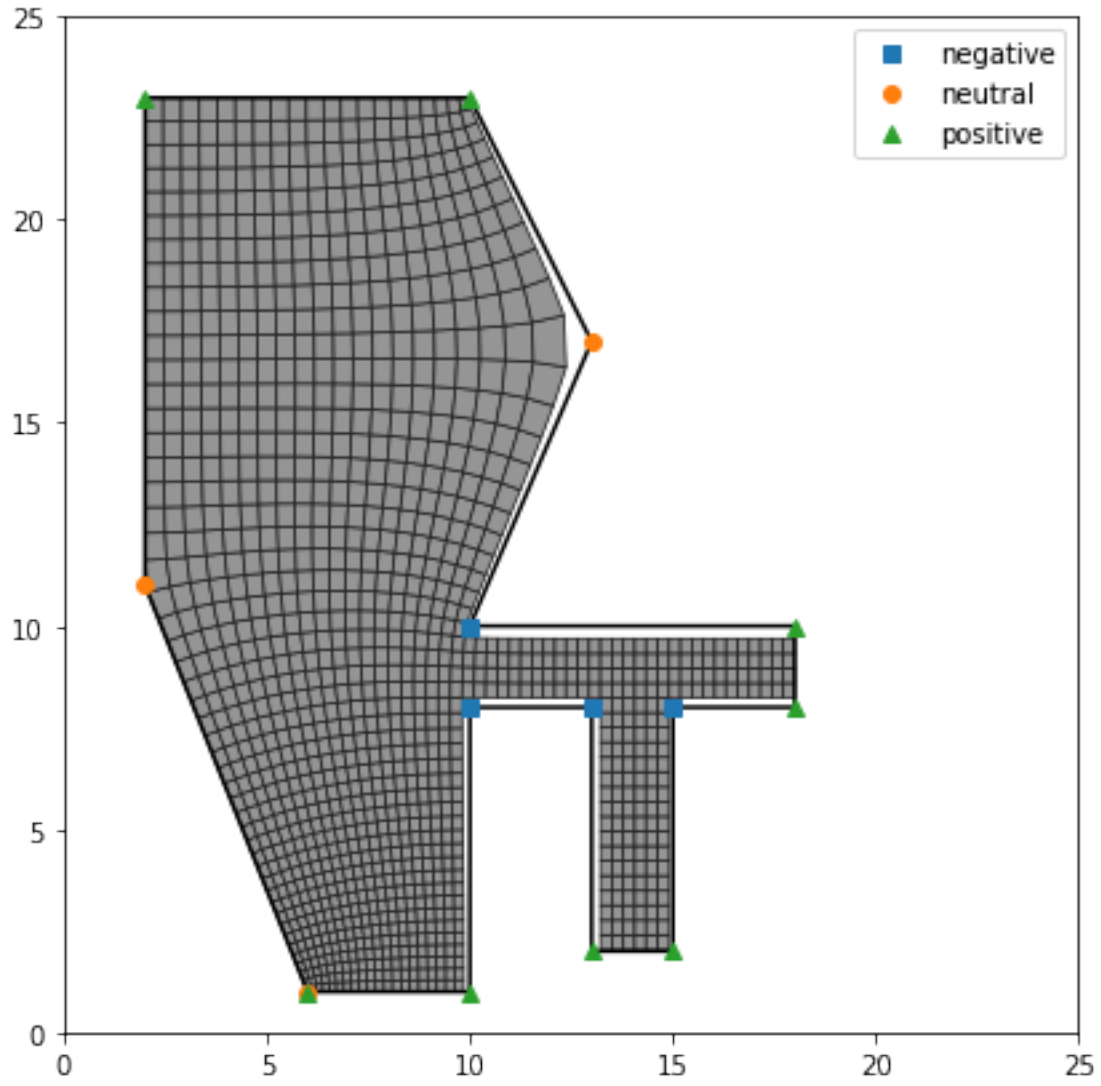
```
In [3]: fig, ax = plt.subplots(figsize=(5, 5), subplot_kw={'aspect':'equal'})
        fig = pgt.viz.plot_domain(domain, betacol='beta', ax=ax)
```



4.1.2 Generating a grid with `pygridgen`, plotting with `pygridtools`

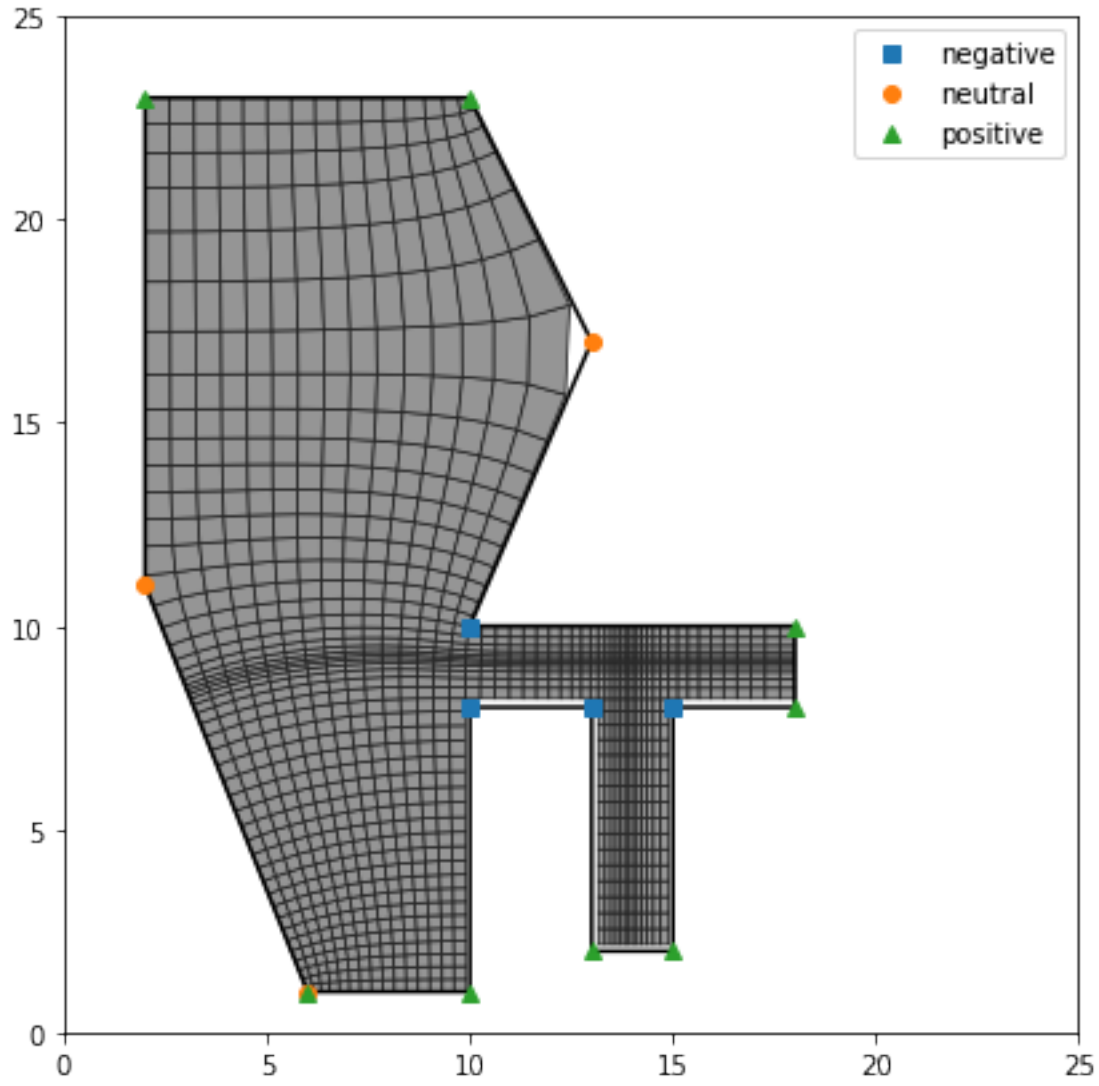
```
In [4]: grid = pgg.Gridgen(domain.geometry.x, domain.geometry.y,
                           domain.beta, shape=(50, 50), ul_idx=2)

        fig, ax = plt.subplots(figsize=(7, 7), subplot_kw={'aspect':'equal'})
        fig = pgt.viz.plot_cells(grid.x, grid.y, ax=ax)
        fig = pgt.viz.plot_domain(domain, betacol='beta', ax=ax)
        ax.set_xlim([0, 25])
        ax.set_ylim([0, 25])
```

```
Out[4]: (0, 25)
```



4.1.3 Using *focus* to refine and coarsen portions of the grid

```
In [5]: focus = pgg.Focus()

        focus.add_focus(0.90, 'y', factor=0.5, extent=0.05)
        focus.add_focus(0.50, 'y', factor=5, extent=0.1)
        focus.add_focus(0.65, 'x', factor=4, extent=0.2)
        grid.focus = focus
        grid.generate_grid()

        fig, ax = plt.subplots(figsize=(7, 7), subplot_kw={'aspect':'equal'})
        fig, cell_artist = pgt.viz.plot_cells(grid.x, grid.y, ax=ax)
        fig, domain_artist = pgt.viz.plot_domain(domain, betacol='beta', ax=ax)
        ax.set_xlim([0, 25])
        ax.set_ylim([0, 25])

Out[5]: (0, 25)
```



4.2 Masking grid cells

This tutorial will demonstrate the following:

1. Basics of grid masking
2. Reading boundary, river, and island data from shapefiles
3. Generating a focused grid
4. Masking land cells from the shapefiles
5. Writing grid data to shapefiles

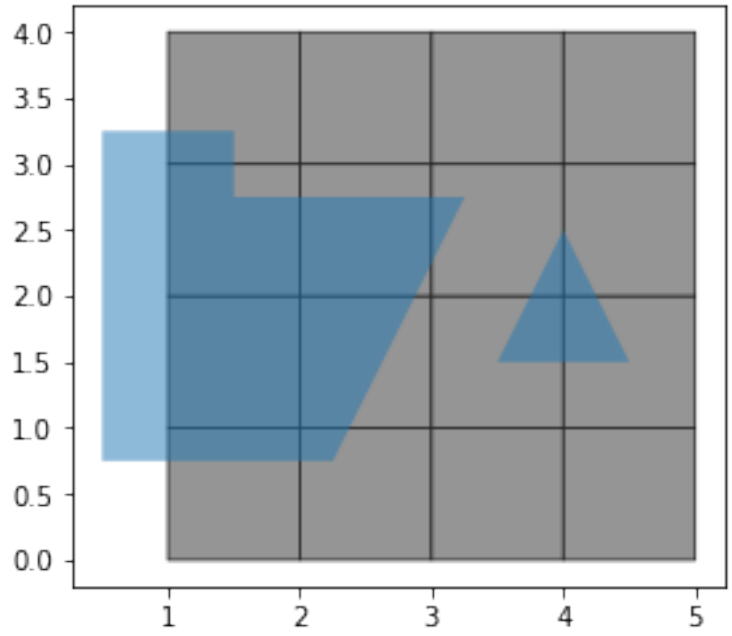
```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import pandas
        from shapely.geometry import Polygon
        import geopandas
```

```
import pygridgen as pgg
import pygridtools as pgt


def show_the_grid(g, domain, river, islands, colors=None):
    fig, (ax1, ax2) = plt.subplots(figsize=(12, 7.5), ncols=2, sharex=True, sharey=True)

    _ = g.plot_cells(ax=ax1, cell_kws=dict(cmap='bone', colors=colors))
    _ = g.plot_cells(ax=ax2, cell_kws=dict(cmap='bone', colors=colors))

    pgt.viz.plot_domain(domain, ax=ax2)
    river.plot(ax=ax2, alpha=0.5, color='C0')
    islands.plot(ax=ax2, alpha=0.5, color='C2')


    _ = ax1.set_title('just the grid')
    _ = ax2.set_title('the grid + all the fixins')

    return fig


def make_fake_bathy(grid):
    j_cells, i_cells = grid.cell_shape
    y, x = np.mgrid[:j_cells, :i_cells]
    z = (y - (j_cells // 2))** 2 - x
    return z
```

4.2.1 Masking basics

Let's consider a simple, orthogonal 5×5 unit grid and a basic rectangle that we will use to mask some elements of the grid:

```
In [2]: y, x = np.mgrid[:5, 1:6]
        mg = pgt.ModelGrid(x, y)

        mask = geopandas.GeoSeries(map(Polygon, [
            [(0.50, 3.25), (1.50, 3.25), (1.50, 2.75),
             (3.25, 2.75), (2.25, 0.75), (0.50, 0.75)],
            [(4.00, 2.50), (3.50, 1.50), (4.50, 1.50)]
        ]))

        fig, ax = plt.subplots()
        fig, cells = mg.plot_cells(ax=ax)
        mask.plot(ax=ax, alpha=0.5)

Out[2]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8e64796908>
```



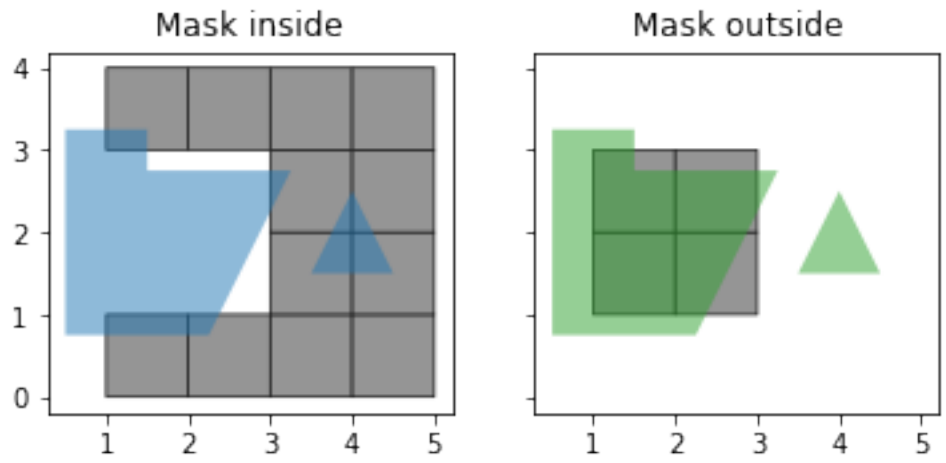
Applying the masks options

You have couple of options when applying a mask to a grid

1. `min_nodes=3` - This parameter configures how manx nodes of a cell must be inside a polygon to flag the whole cell as inside thet polygon.
2. `use_existing=True` - When this is `True` the new mask determined from the passed polygons will be unioned (`np.bitwise_or`) with anx existing mask that may be present. When this is `False` the old mask is completely overwritten with the new mask.

Masking inside vs outside a polygon

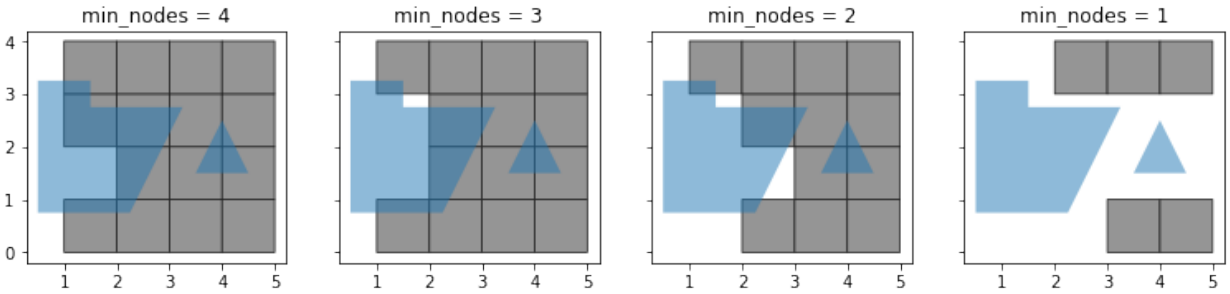
```
In [3]: fig, (ax1, ax2) = plt.subplots(figsize=(6, 3), ncols=2, sharex=True, sharey=True)

        common_opts = dict(use_existing=False)

        # mask inside
        _ = (
            mg.mask_centroids(inside=mask, **common_opts)
              .plot_cells(ax=ax1)
        )
        mask.plot(ax=ax1, alpha=0.5, color='C0')
        ax1.set_title('Mask inside')

        # mask outside
        _ = (
            mg.mask_centroids(outside=mask, **common_opts)
              .plot_cells(ax=ax2)
        )
        mask.plot(ax=ax2, alpha=0.5, color='C2')
        _ = ax2.set_title("Mask outside")
```



Masking with nodes instead of centroids

This time, we'll mask with the nodes of the cells instead of the centroids. We'll show four different masks, each generated with a different minimum number of nodes requires to classify a cell as inside the polygon.

```
In [4]: fig, axes = plt.subplots(figsize=(13, 3), ncols=4, sharex=True, sharey=True)


        common_opts = dict(use_existing=False)

        for ax, min_nodes in zip(axes.flat, [4, 3, 2, 1]):
            # mask inside
            _ = (
                mg.mask_nodes(inside=mask, min_nodes=min_nodes, **common_opts)
                  .plot_cells(ax=ax)
            )
            mask.plot(ax=ax, alpha=0.5)
            ax.set_title("min_nodes = {:d}".format(min_nodes))
```

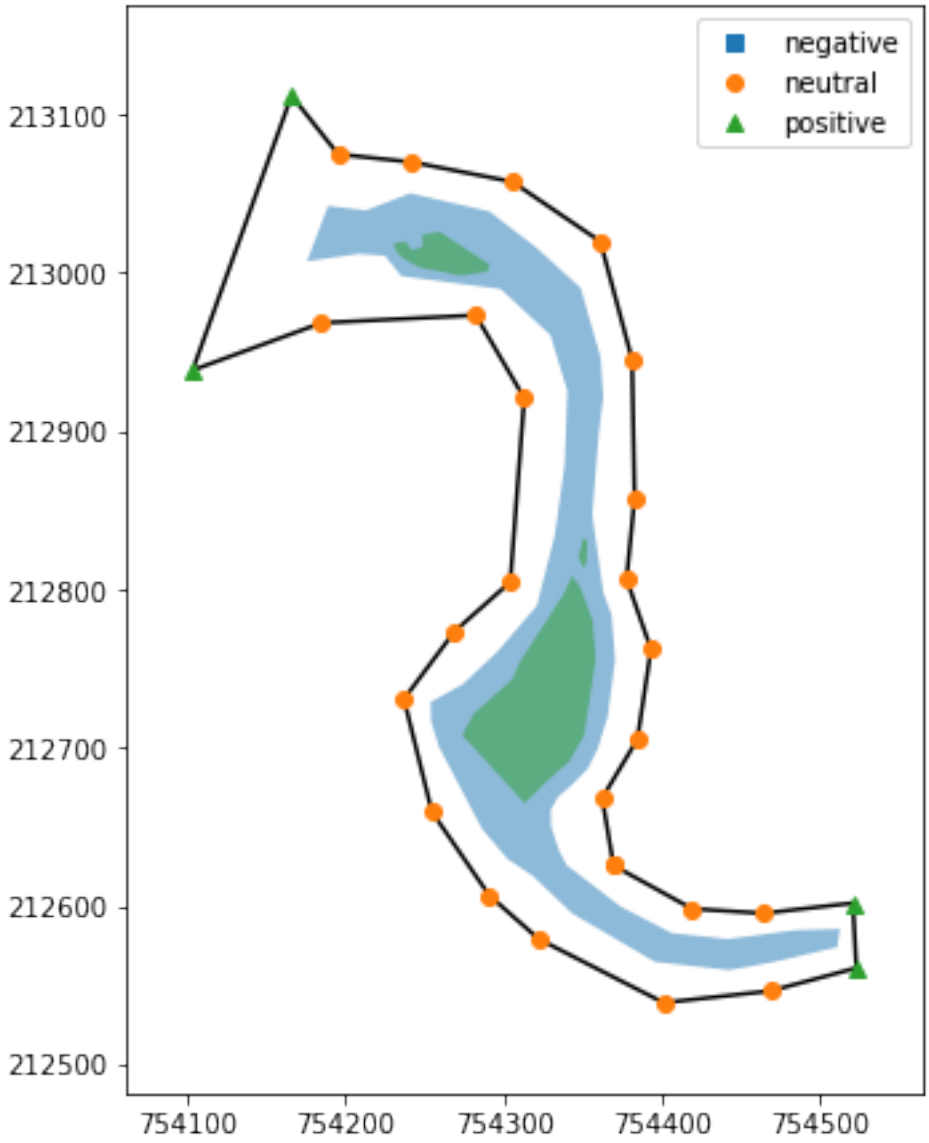


4.2.2 Example with islands and rivers

```
In [5]: domain = (
            geopandas.read_file("masking_data/input/GridBoundary.shp")
                .sort_values(by=['sort_order'])
        )

        river = geopandas.read_file("masking_data/input/River.shp")
```

```
islands = geopandas.read_file("masking_data/input/Islands.shp")


fig, ax = plt.subplots(figsize=(7.5, 7.5), subplot_kw={'aspect': 'equal'})
fig = pgt.viz.plot_domain(domain, betacol='beta', ax=ax)
river.plot(ax=ax, color='C0', alpha=0.5)
islands.plot(ax=ax, color='C2', alpha=0.5)
```

Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8e1d2d3f28>



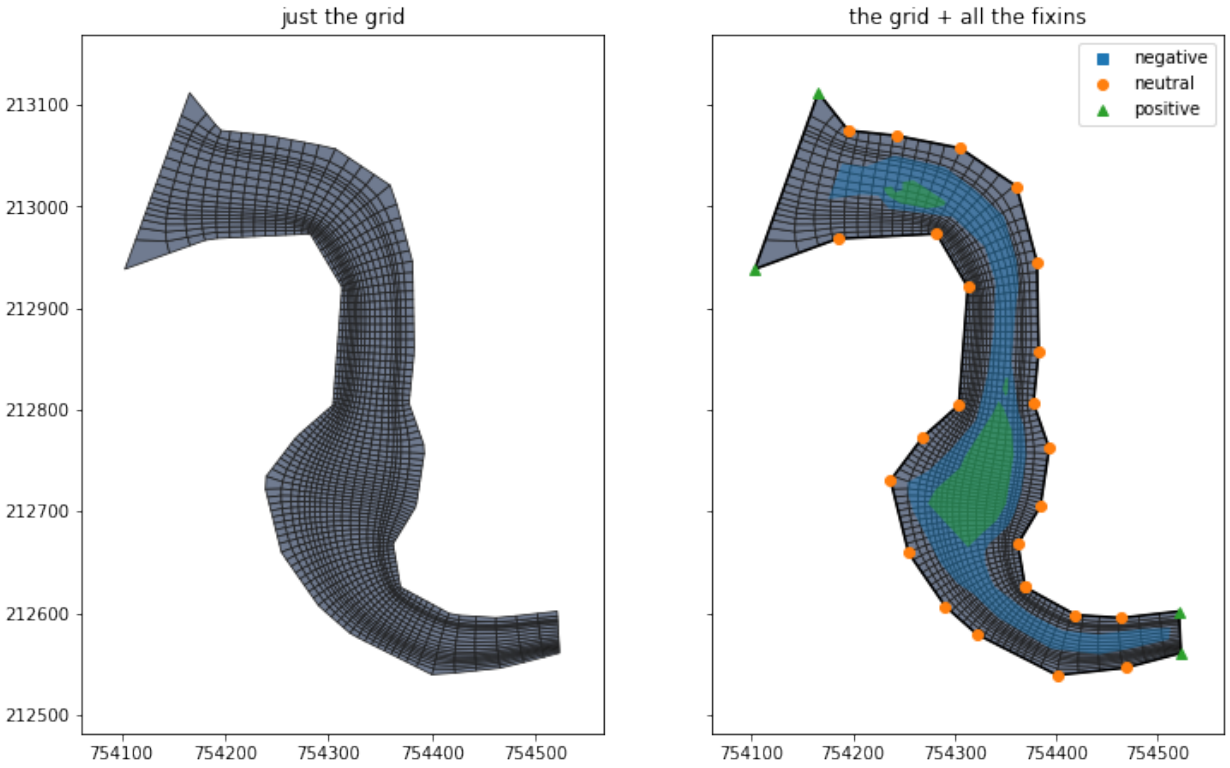
Creating a `Gridgen` objects

```
In [6]: # number of nodes in each dimension
        i_nodes = 100
        j_nodes = 20

        # grid focus
```

```
focus = pgg.Focus()

# tighten the grid in the channels around the big island
focus.add_focus(5. / j_nodes, 'y', 4., extent=8./j_nodes)
focus.add_focus(14.5 / j_nodes, 'y', 4., extent=4./j_nodes)

# coarsen the grid upstream
focus.add_focus(98. / i_nodes, 'x', 0.25, extent=4./i_nodes)

# tighten the grid around the big island's bend
focus.add_focus(52. / i_nodes, 'x', 4., extent=20./i_nodes)

# generate the main grid
grid = pgt.make_grid(
    domain=domain,
    ny=j_nodes,
    nx=i_nodes,
    ul_idx=17,
    focus=focus,
    rawgrid=False
)
```

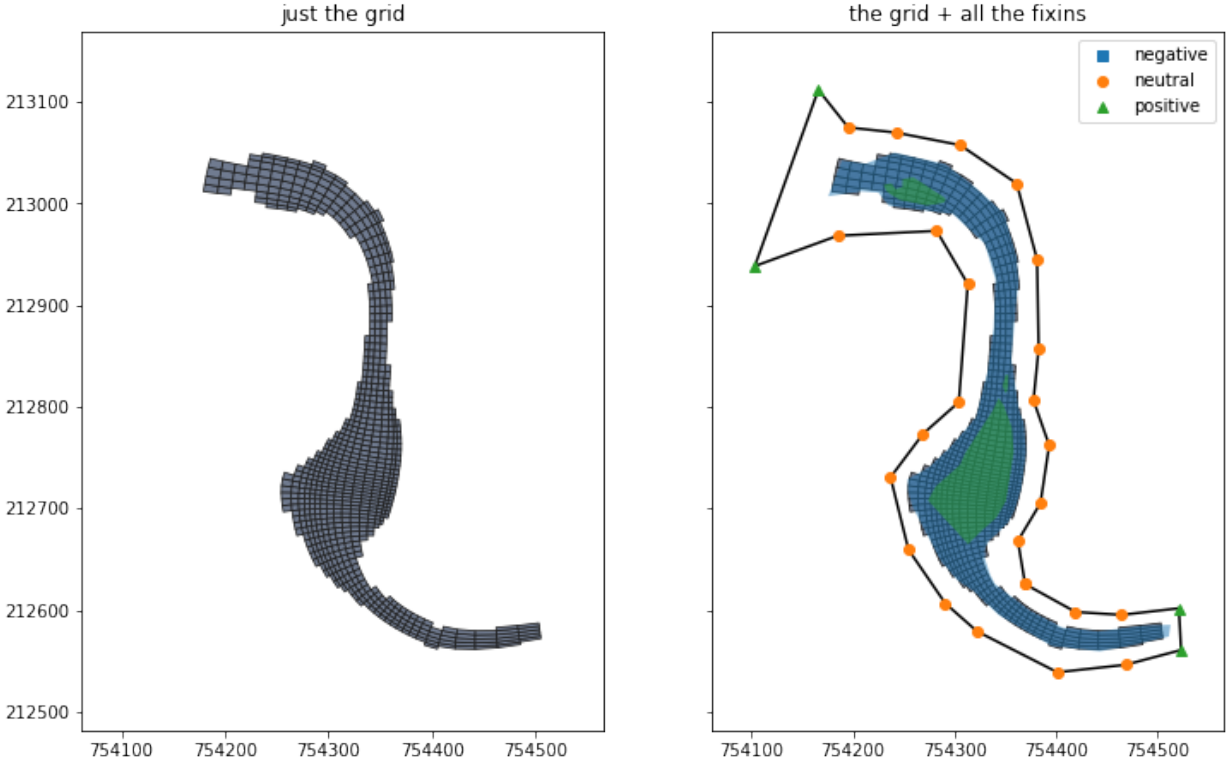
Show the raw (unmasked) grid

```
In [7]: fig = show_the_grid(grid, domain, river, islands)
```



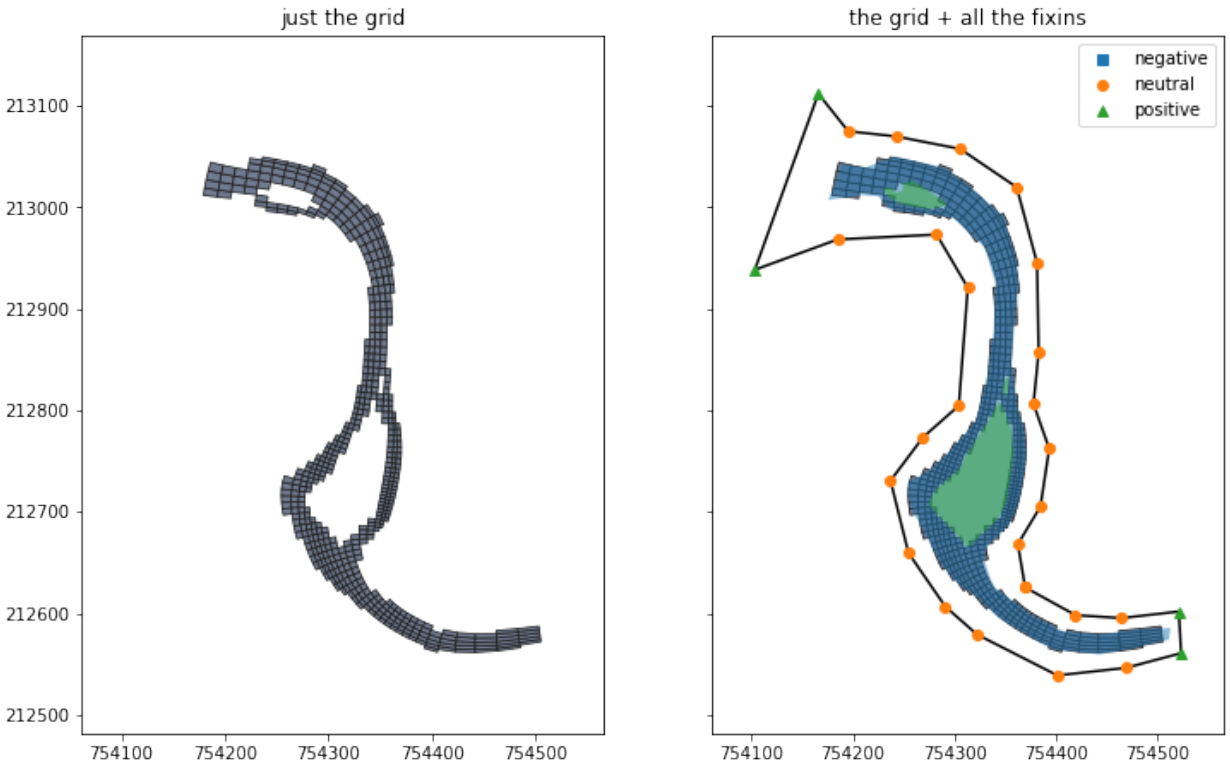
Mask out everything beyond the river banks

```
In [8]: masked_river = grid.mask_centroids(outside=river)
        fig = show_the_grid(masked_river, domain, river, islands)
```



Loop through and mask out the islands

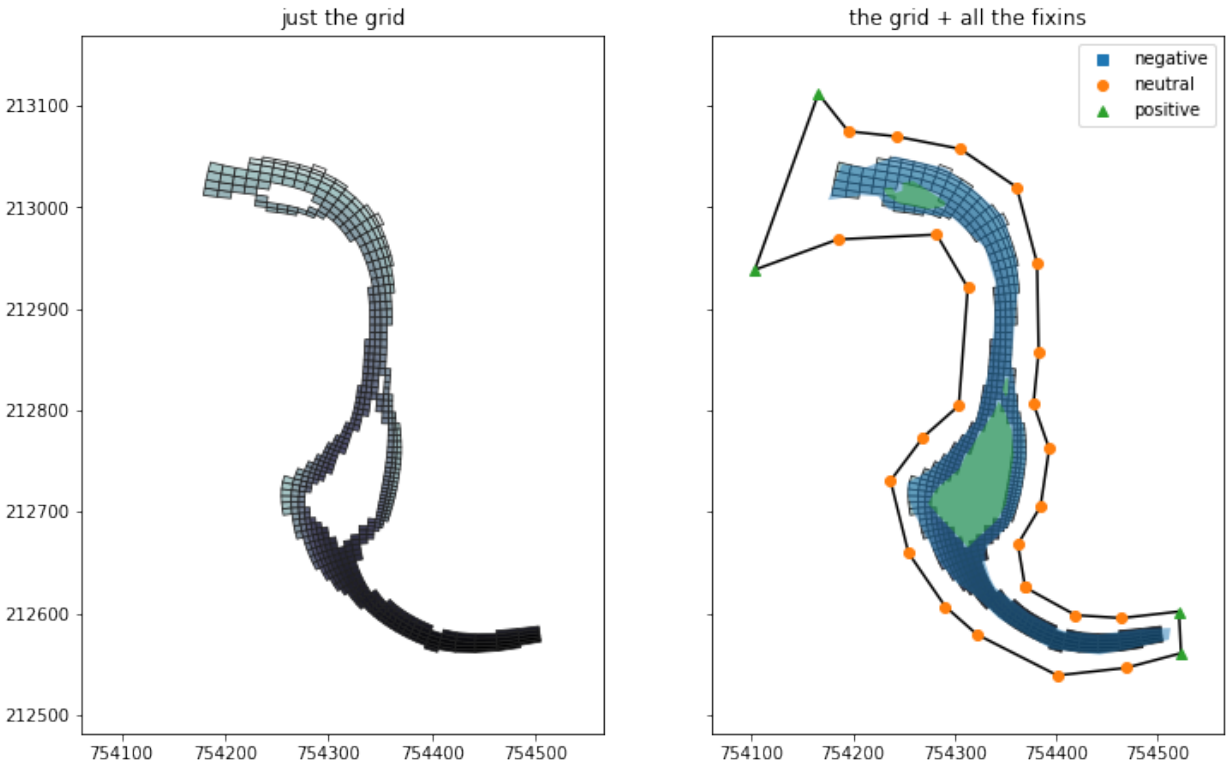
```
In [9]: # inside the multiple islands
        masked_river_islands = masked_river.mask_centroids(inside=islands)
        fig = show_the_grid(masked_river_islands, domain, river, islands)
```



4.2.3 Plotting with e.g., bathymetry data

The key here is that you need an array that is the same shape as the centroids of your grid

```
In [10]: fake_bathy = make_fake_bathy(masked_river_islands)
         fig = show_the_grid(masked_river_islands,  domain, river, islands, colors=fake_bathy)
```



4.2.4 Exporting the masked cells to a GIS file

```
In [11]: gdf = masked_river_islands.to_polygon_geodataframe(usemask=True)
         gdf.to_file('masking_data/output/ModelCells.shp')
```

4.2.5 View the final input and output in the QGIS file in `examples/masking_data/Grid.qgs`

4.3 Grid Manipulations (merge, split, refine, transform)

4.3.1 Notes

Most grid transformations such as `merge` and `transpose` return a new object, allowing consecutive operations to be chained together. Optionally, you can pass `inplace=True` to the call signature to modify the existing object and return `None`. Both approaches are demonstrated below.

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import pandas
        from shapely.geometry import Point, Polygon
        import geopandas

        import pygridgen as pgg
        import pygridtools as pgt
```

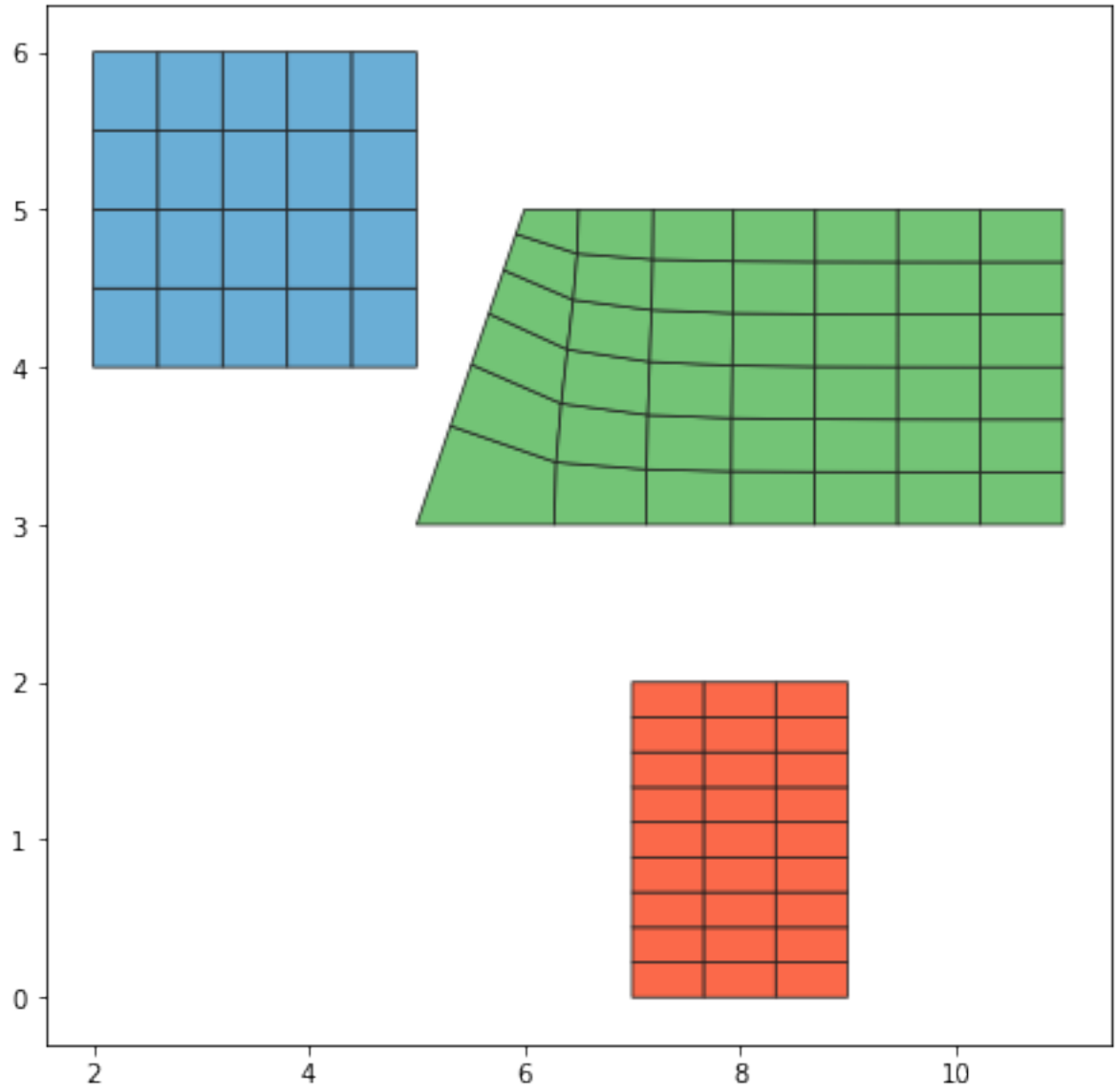
4.3.2 Basic merging operations

The function below create our 3 test model grids moving counter-clockwise in the figure shown two cells down.

```
In [2]: def to_gdf(df):

            return (
                df.assign(geometry=df.apply(lambda r: Point(r.x, r.y), axis=1))
                  .drop(columns=['x', 'y'])
                  .pipe(geopandas.GeoDataFrame)
            )


        def make_test_grids():
            domain1 = pandas.DataFrame({'x': [2, 5, 5, 2], 'y': [6, 6, 4, 4], 'beta': [1, 1, 1, 1]})
            domain2 = pandas.DataFrame({'x': [6, 11, 11, 5], 'y': [5, 5, 3, 3], 'beta': [1, 1, 1, 1]}
            domain3 = pandas.DataFrame({'x': [7, 9, 9, 7], 'y': [2, 2, 0, 0], 'beta': [1, 1, 1, 1]})

            grid1 = pgt.make_grid(domain=to_gdf(domain1), nx=6, ny=5, rawgrid=False)
            grid2 = pgt.make_grid(domain=to_gdf(domain2), nx=8, ny=7, rawgrid=False)
            grid3 = pgt.make_grid(domain=to_gdf(domain3), nx=4, ny=10, rawgrid=False)

            return grid1, grid2, grid3
```

Display positions of grids relative to each other

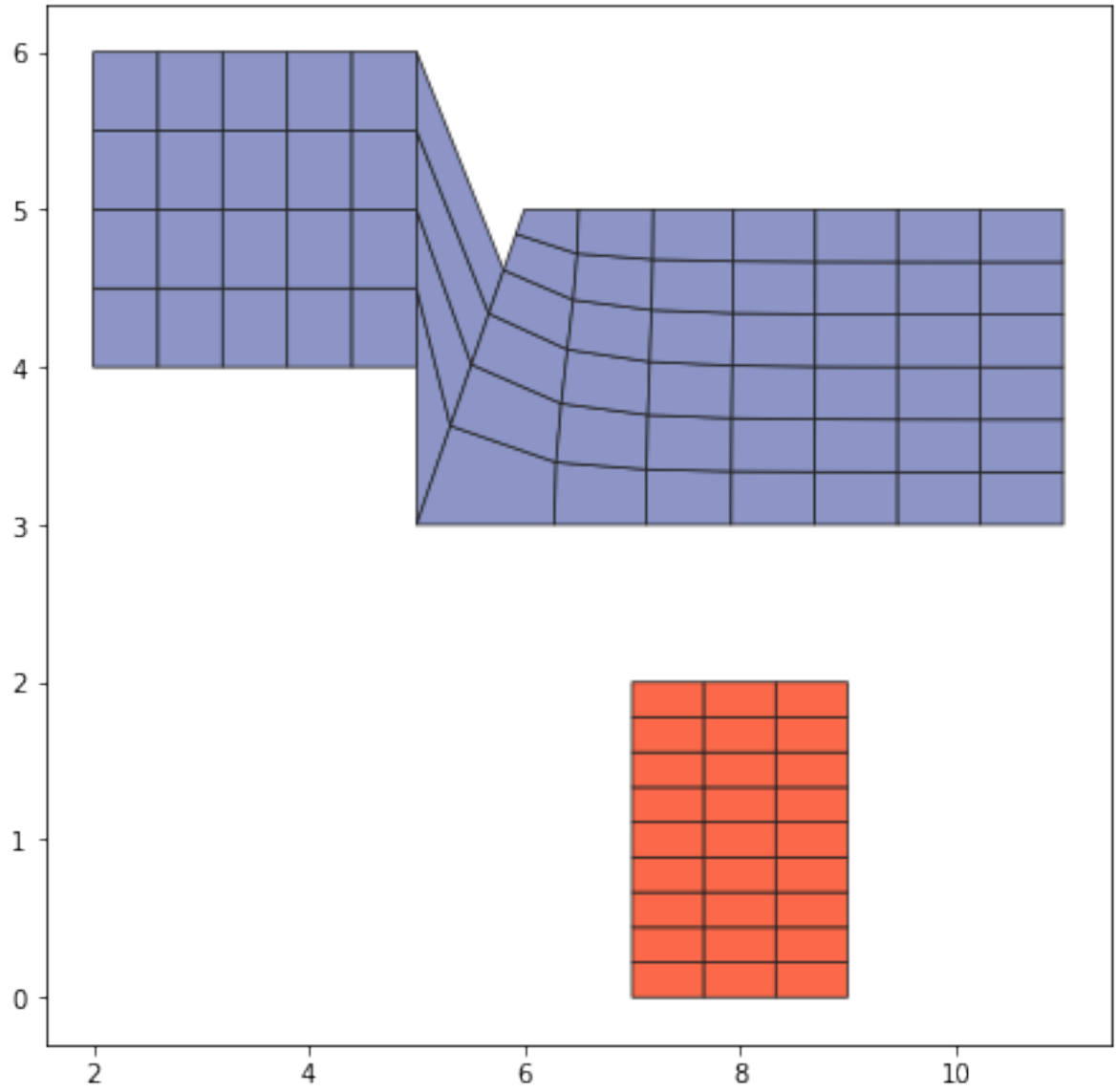
```
In [3]: grid1, grid2, grid3 = make_test_grids()

        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = grid1.plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
        _ = grid2.plot_cells(ax=ax, cell_kws=dict(cmap='Greens'))
        _ = grid3.plot_cells(ax=ax, cell_kws=dict(cmap='Reds'))
```



Merge grids 1 and 2 together, horizontally

By default, the bottom rows are aligned and the cell mask is not updated. We do that manually for now.

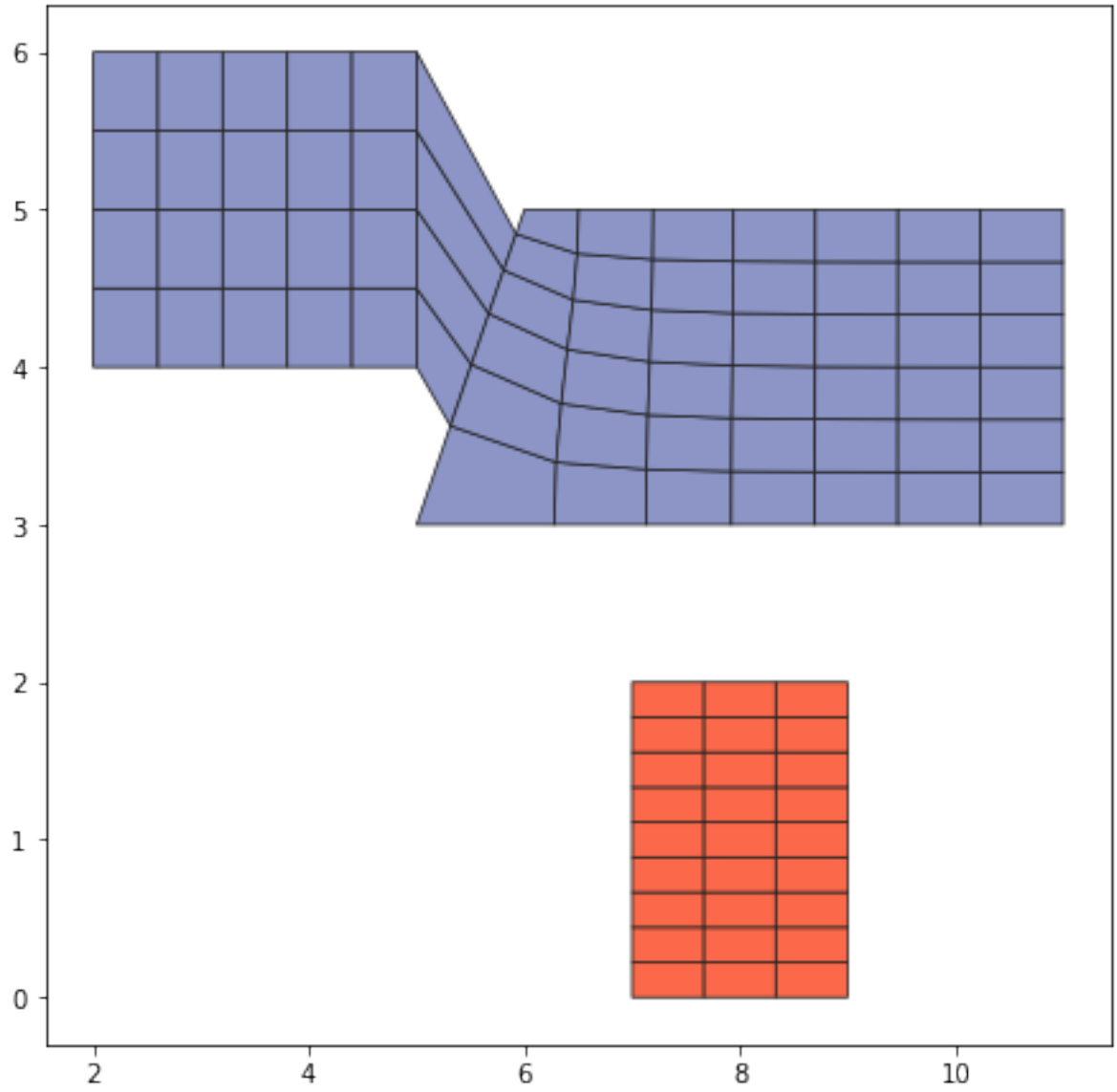
```
In [4]: one_two = grid1.merge(grid2, how='horiz')
        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = one_two.plot_cells(ax=ax, cell_kws=dict(cmap='BuPu'))
        _ = grid3.plot_cells(ax=ax, cell_kws=dict(cmap='Reds'))
```



Use the shift parameter to center grid 2

Use `shift=-1` since we're sliding grid 2's i-j indexes downward relative to grid 1

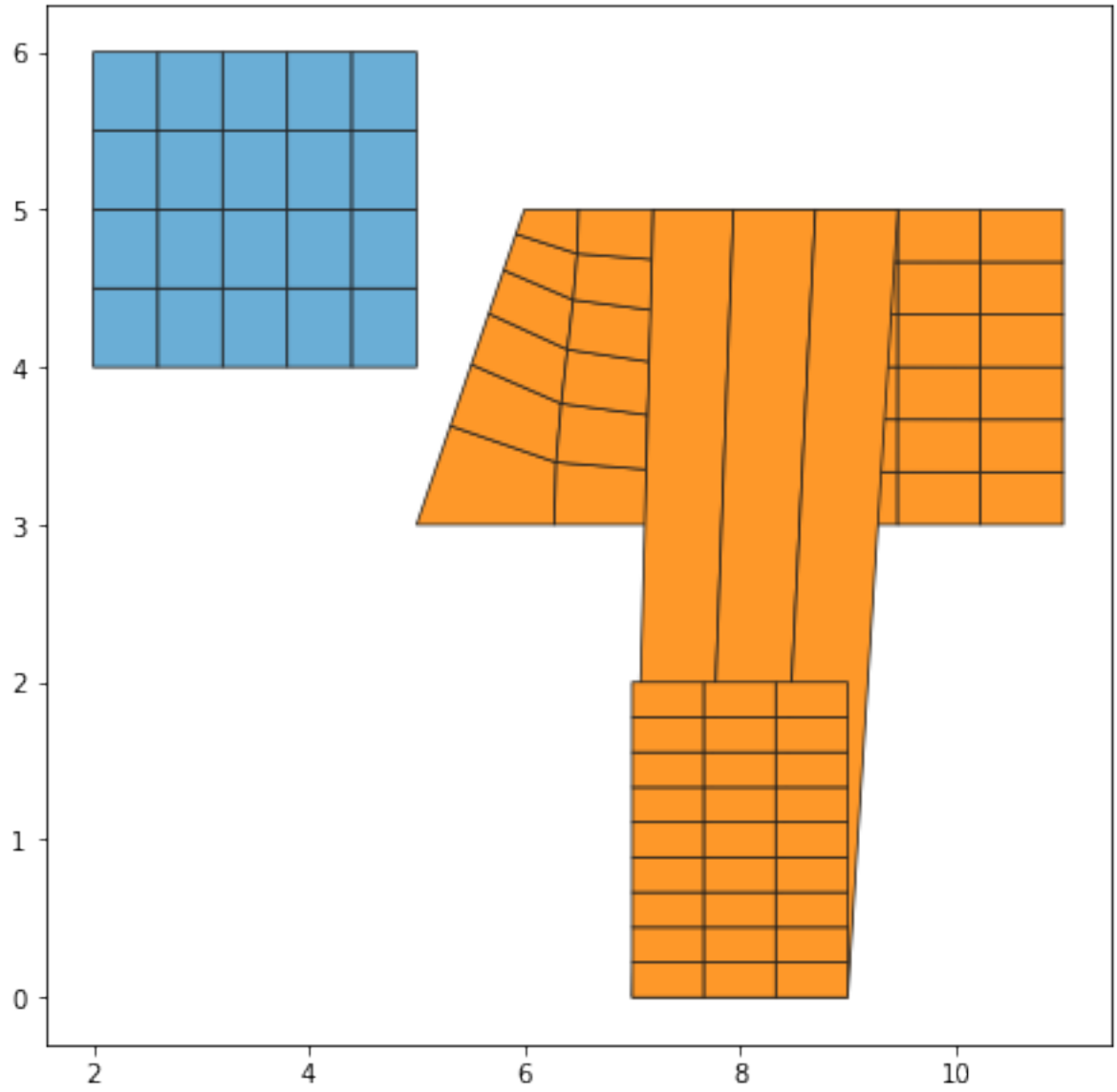
```
In [5]: one_two = grid1.merge(grid2, how='horiz', shift=-1)

        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = one_two.plot_cells(ax=ax, cell_kws=dict(cmap='BuPu'))
        _ = grid3.plot_cells(ax=ax, cell_kws=dict(cmap='Reds'))
```



Vertically merge grid 2 and grid 3

Notice that by default, the grids are left-aligned and the *bottom* of grid 3 ties into the *top* of grid 2

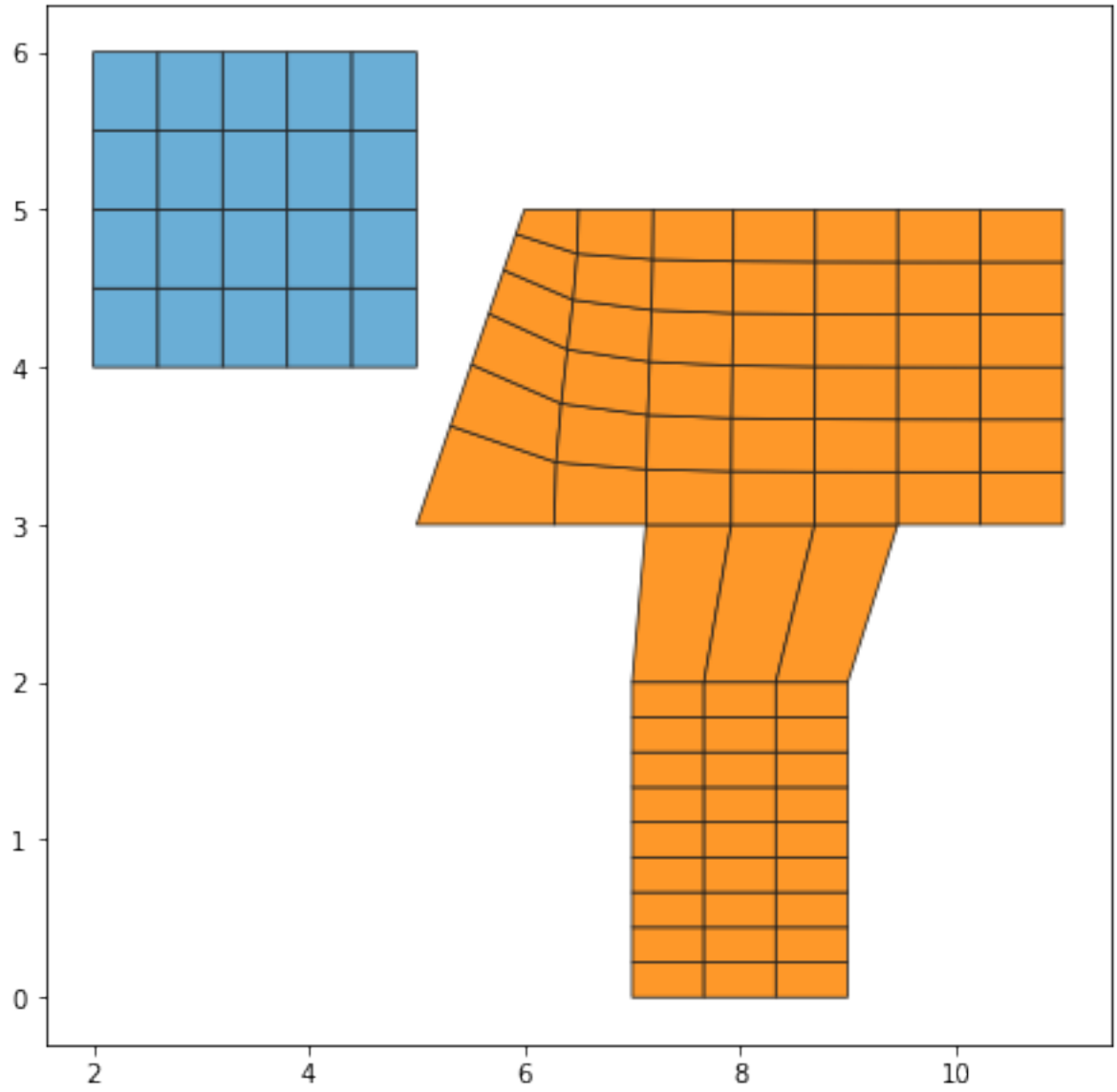
```
In [6]: two_three = grid2.merge(grid3, how='vert', shift=2)
        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = grid1.plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
        _ = two_three.plot_cells(ax=ax, cell_kws=dict(cmap='YlOrBr'))
```



Try again, switching the order of the grids

Notice the change in sign of the `shift` parameter.

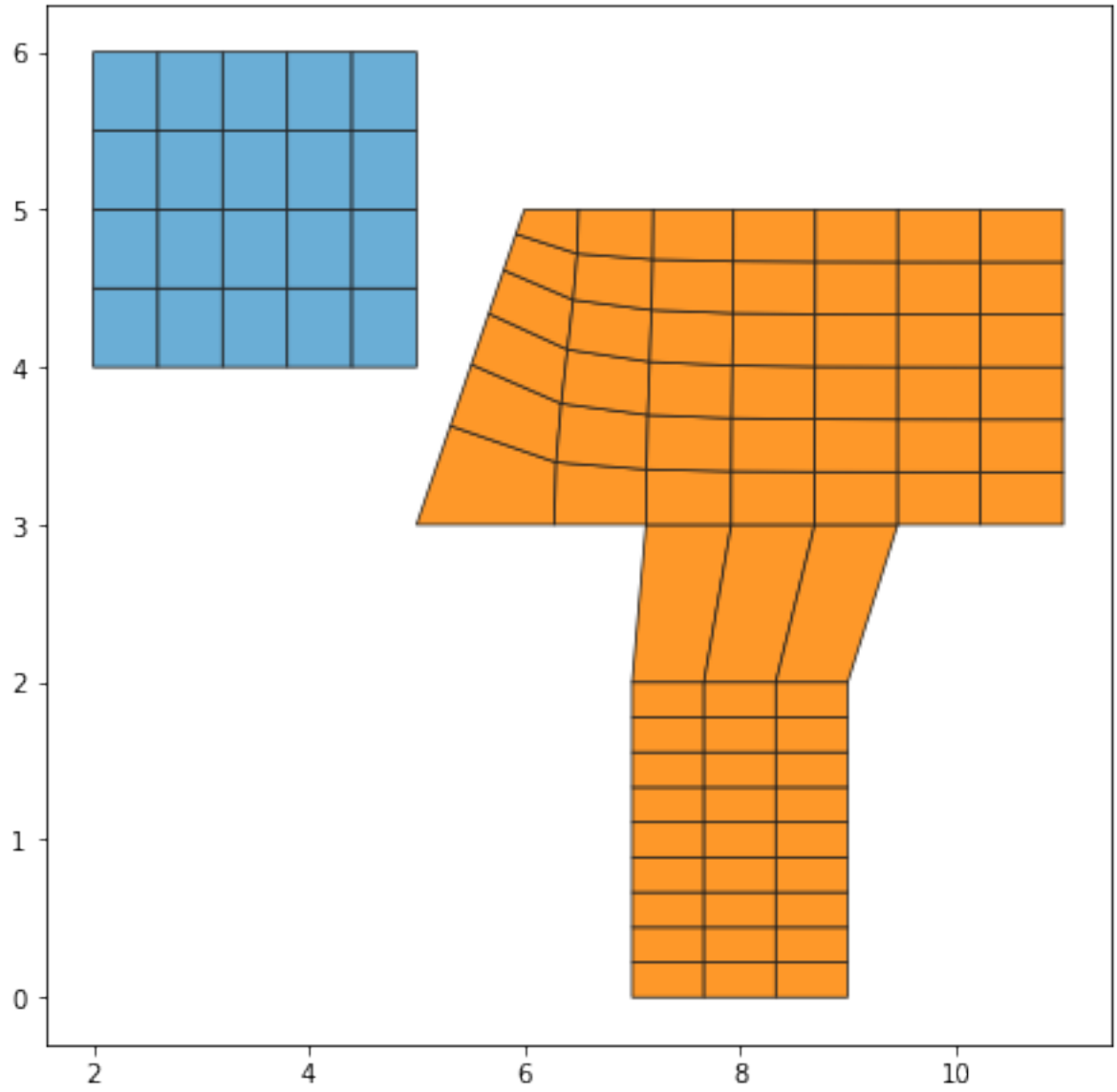
```
In [7]: two_three = grid3.merge(grid2, how='vert', shift=-2)

        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = grid1.plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
        _ = two_three.plot_cells(ax=ax, cell_kws=dict(cmap='YlOrBr'))
```



Alternatively, you can switch the arguments and use `where='-'` to indicate that the "other" grid is below the first.

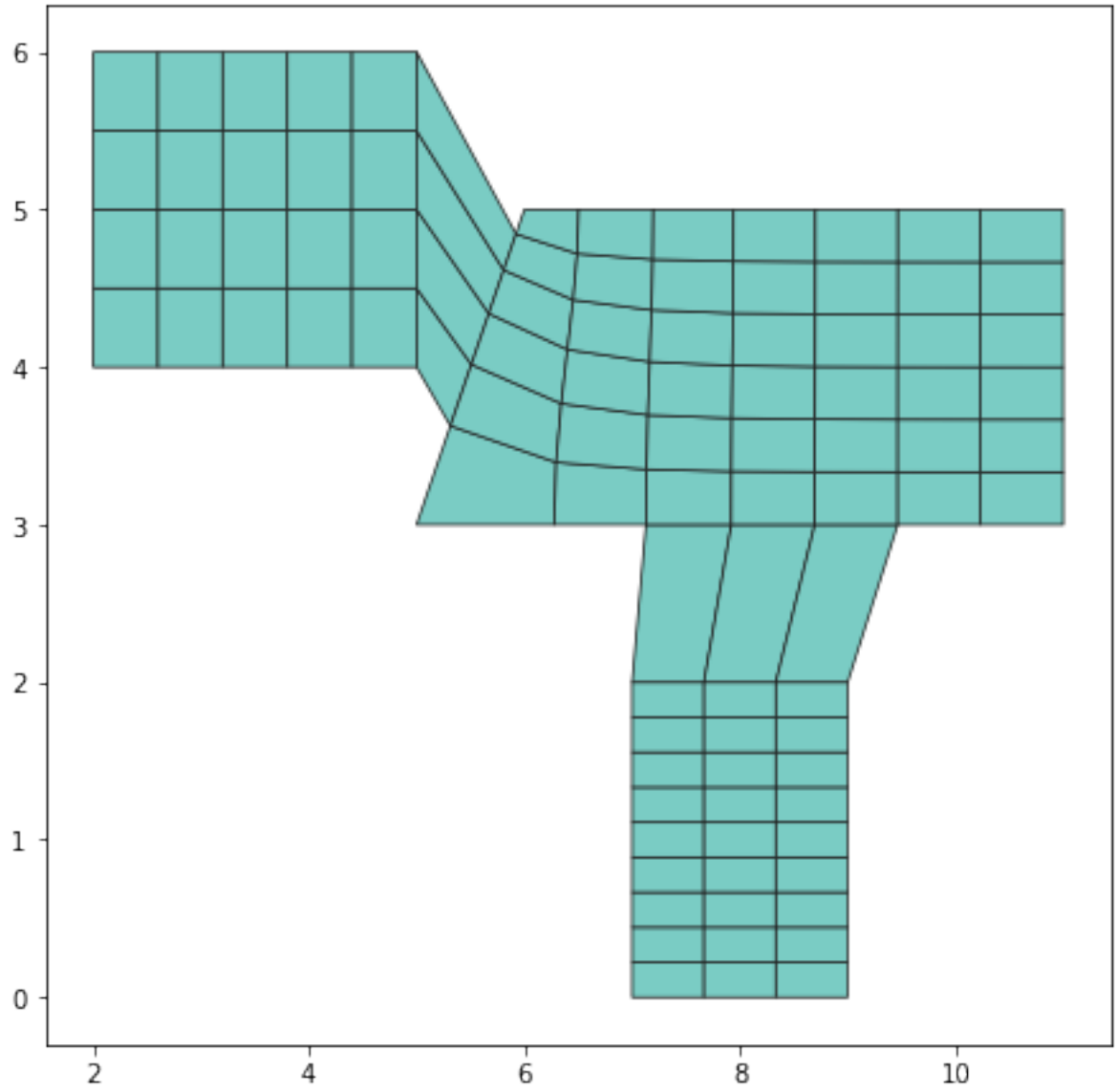
And the sign of the `shift` parameter returns to its original value.

```
In [8]: two_three = grid2.merge(grid3, how='vert', where='-', shift=2)

        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = grid1.plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
        _ = two_three.plot_cells(ax=ax, cell_kws=dict(cmap='YlOrBr'))
```



Now merge all three in a single chained operation (`inplace=False`).

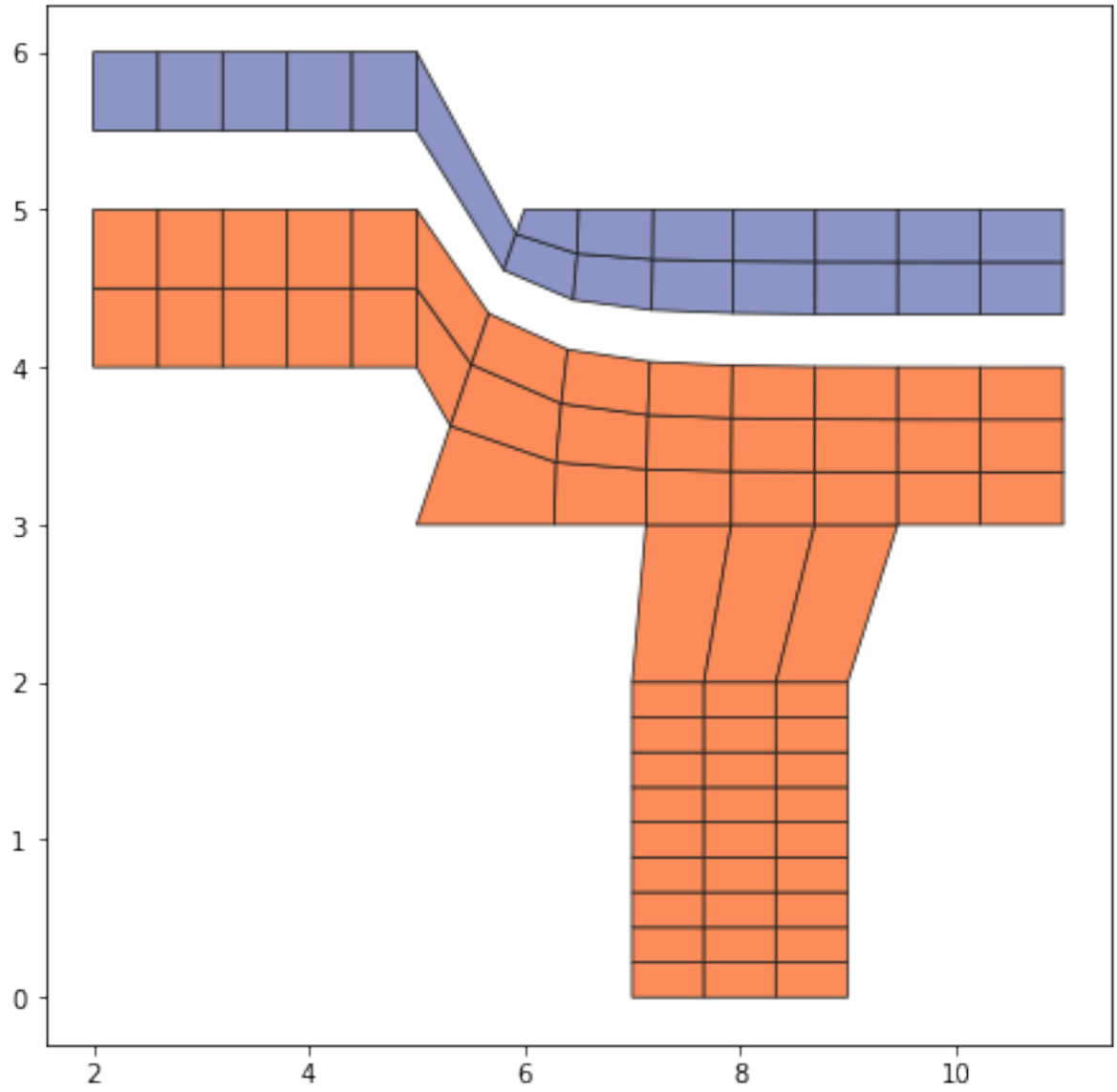
```
In [9]: grid1, grid2, grid3 = make_test_grids()
        all_grids = (
            grid2.merge(grid3, how='vert', where='-', shift=2)
                 .merge(grid1, how='horiz', where='-', shift=11)
        )

        fig, ax = plt.subplots(figsize=(7.5, 7.5))
        _ = all_grids.plot_cells(ax=ax, cell_kws=dict(cmap='GnBu'))
```



4.3.3 Split the final grid into two vertical parts

```
grid.split(<index of split>, axis=0)
```

```
In [10]: grid_bottom, grid_top = all_grids.split(14, axis=0)

         fig, ax = plt.subplots(figsize=(7.5, 7.5))
         _ = grid_bottom.plot_cells(ax=ax, cell_kws=dict(cmap='OrRd'))
         _ = grid_top.plot_cells(ax=ax, cell_kws=dict(cmap='BuPu'))
```

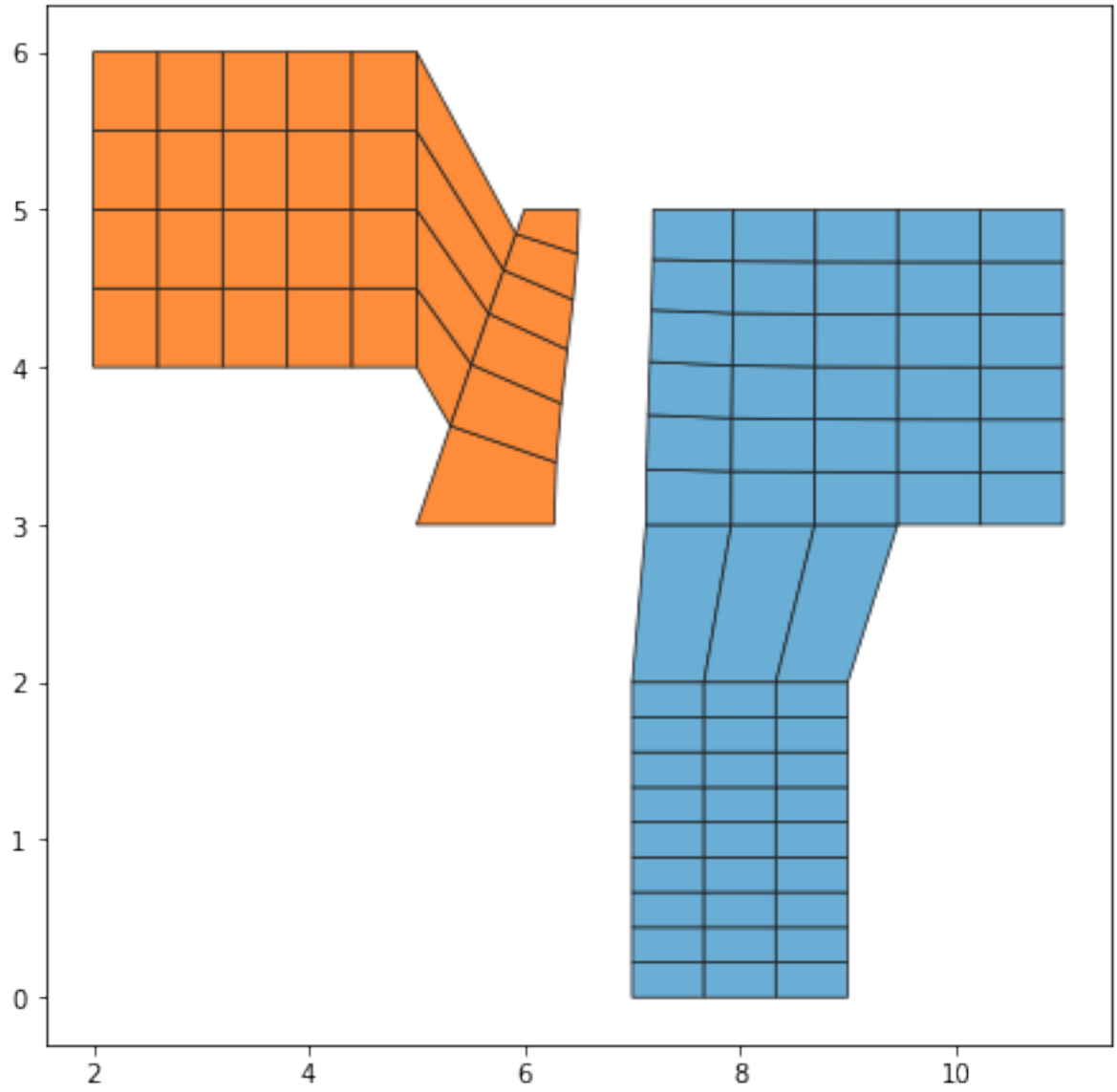


4.3.4 Splitting and linearly refining columns and rows

Split the final grid into two horizontal parts

```
grid.split(<index of split>, axis=1)
```

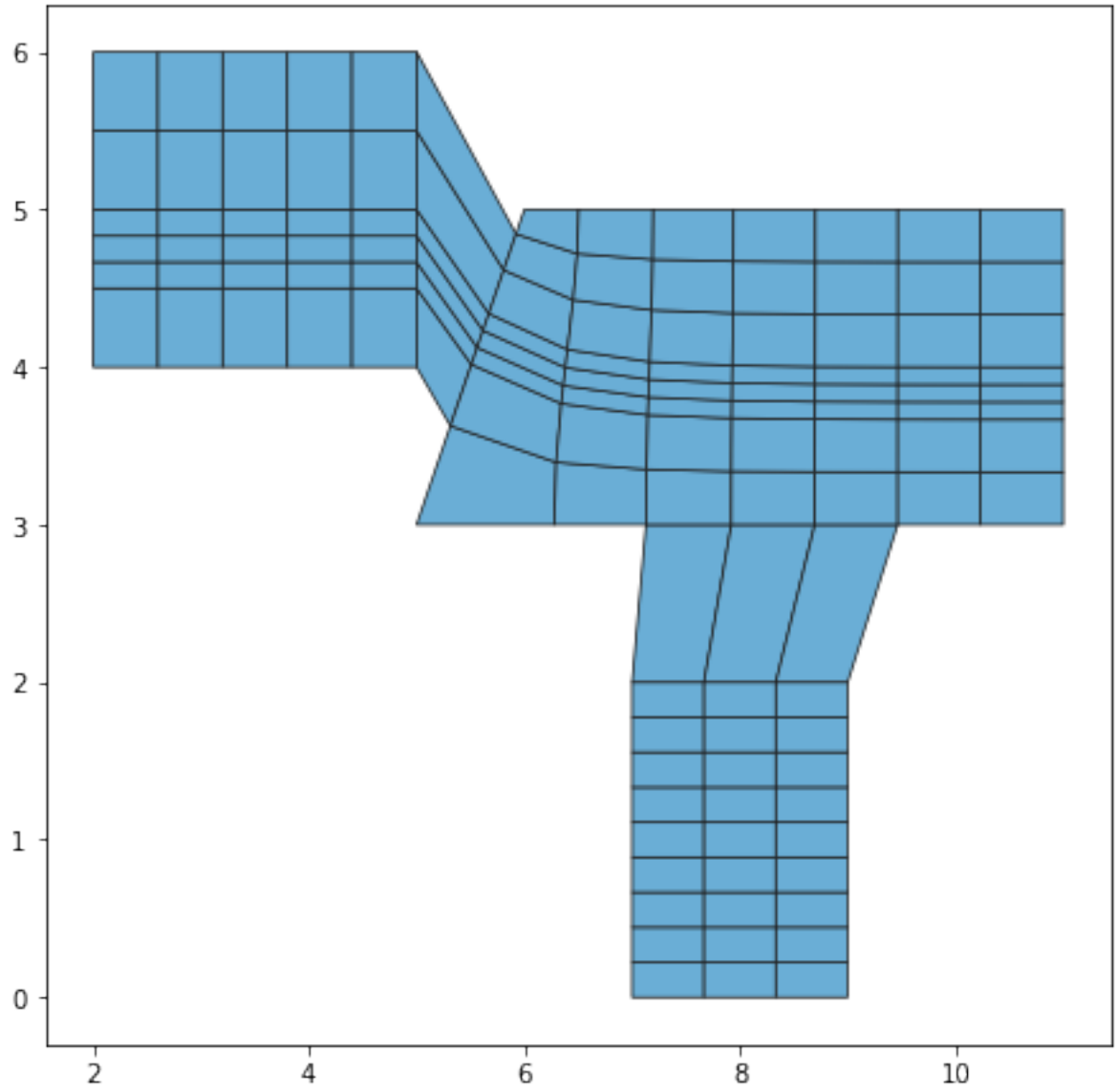
```
In [11]: grid_left, grid_right = all_grids.split(8, axis=1)

         fig, ax = plt.subplots(figsize=(7.5, 7.5))
         _ = grid_left.plot_cells(ax=ax, cell_kws=dict(cmap='Oranges'))
         _ = grid_right.plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
```



Refine individual rows of the grid cells

```
grid.refine(<index of cell>, axis=0, n_points=<num. of divisions>)
```

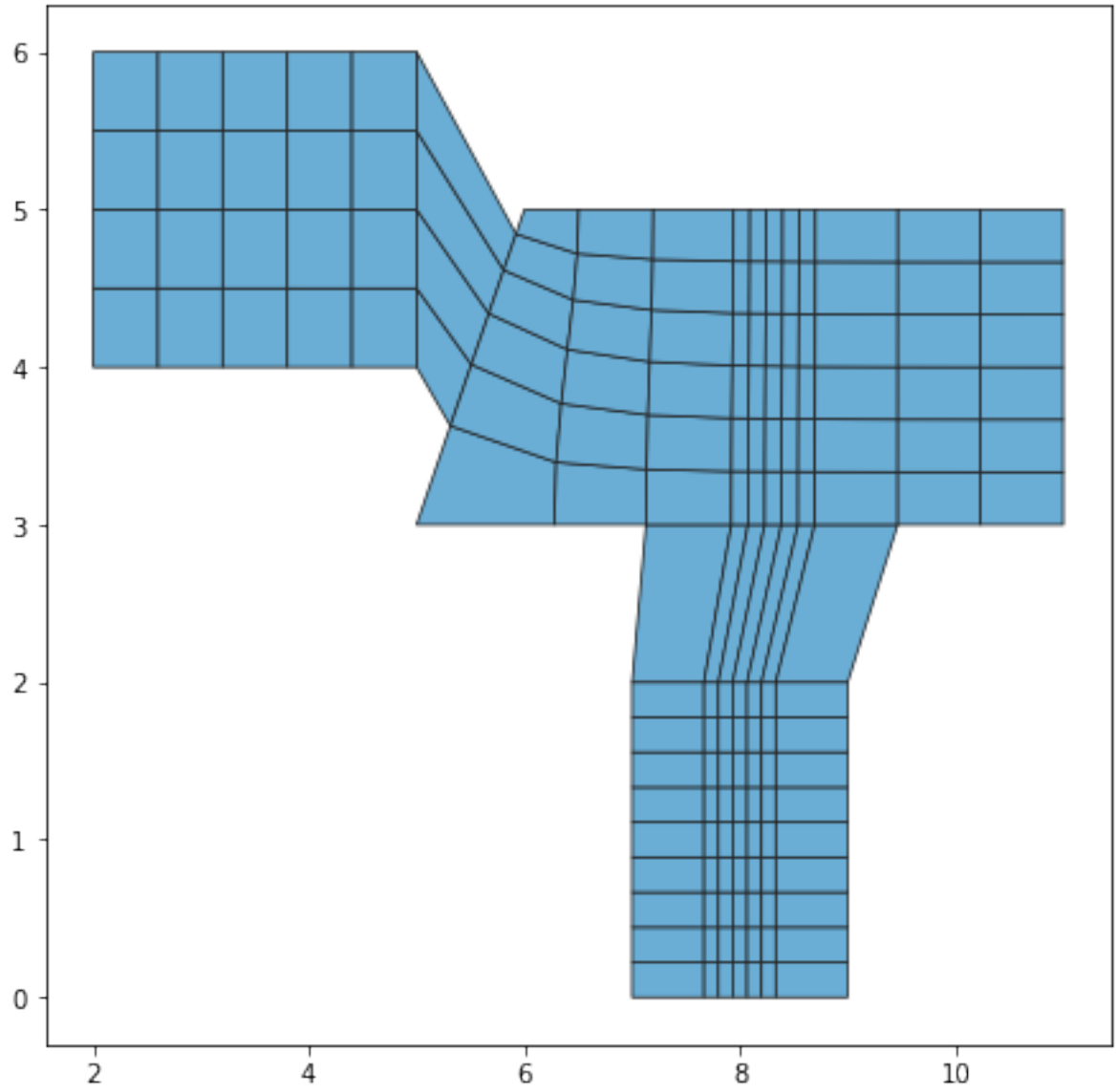
```
In [12]: fig, ax = plt.subplots(figsize=(7.5, 7.5))
         _ = (
             all_grids
                 .insert(13, axis=0, n_nodes=2)
                 .plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
         )
```



Refine individual columns of the grid cells

```
grid.refine(<index of cell>, axis=1, n_points=<num. of divisions>)
```

```
In [13]: fig, ax = plt.subplots(figsize=(7.5, 7.5))
         _ = (
             all_grids
                 .insert(10, axis=1, n_nodes=4)
                 .plot_cells(ax=ax, cell_kws=dict(cmap='Blues'))
         )
```

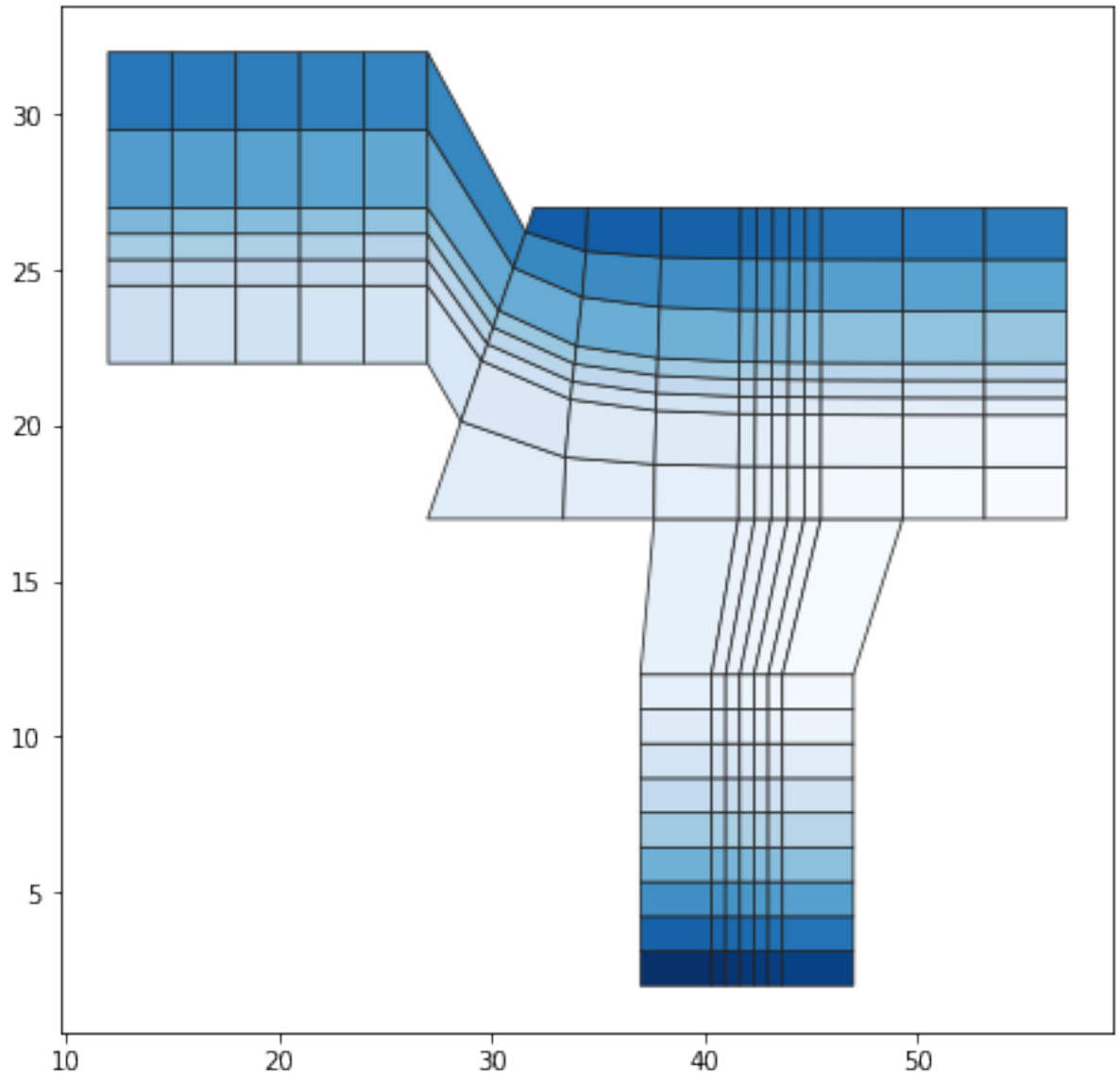


4.3.5 Chained operations

One big chained operation for fun

```
In [14]: def make_fake_bathy(grid):
             j_cells, i_cells = grid.cell_shape
             y, x = np.mgrid[:j_cells, :i_cells]
             z = (y - (j_cells // 2))** 2 - x
             return z


         fig, ax = plt.subplots(figsize=(7.5, 7.5))

         g = (
           grid2.merge(grid3, how='vert', where='-', shift=2)
                .merge(grid1, how='horiz', where='-', shift=11)
```

```
        .insert(10, axis=1, n_nodes=4)
        .insert(13, axis=0, n_nodes=2)
        .transform(lambda x: x*5 + 2)
)

bathy = make_fake_bathy(g)
_ = g.plot_cells(ax=ax, cell_kws=dict(cmap='Blues', colors=bathy))
```



CHAPTER 5

API Reference

5.1 The *core* API

pygridtools.core.**transform**(*nodes*, *fxn*, **args*, ***kwargs*)

Apply an arbitrary function to an array of node coordinates.

Parameters

nodes [numpy.ndarray] An N x M array of individual node coordinates (i.e., the x-coords or the y-coords only)

fxn [callable] The transformation to be applied to the whole `nodes` array

args, kwargs Additional positional and keyword arguments that are passed to `fxn`. The final call will be `fxn(nodes, *args, **kwargs)`.

Returns

transformed [numpy.ndarray] The transformed array.

pygridtools.core.**split**(*nodes*, *index*, *axis=0*)

Split a array of nodes into two separate, non-overlapping arrays.

Parameters

nodes [numpy.ndarray] An N x M array of individual node coordinates (i.e., the x-coords or the y-coords only)

index [int] The leading edge of where the split should occur.

axis [int, optional] The axis along which `nodes` will be split. Use *axis = 0* to split along rows and *axis = 1* for columns.

Returns

n1, n2 [numpy.ndarrays] The two non-overlapping sides of the original array.

Raises

ValueError Trying to split `nodes` at the edge (i.e., resulting in the original array and an empty array) will raise an error.

`pygridtools.core.`**`merge`**(*nodes*, *other_nodes*, *how='vert'*, *where='+'*, *shift=0*)

Merge two sets of nodes together.

Parameters

nodes, other_nodes [numpy.ndarrays] The sets of nodes that will be merged.

how [string, optional (default = 'vert')] The method through wich the arrays should be stacked. *'Vert'* is analogous to *numpy.vstack*. *'Horiz'* maps to *numpy.hstack*.

where [string, optional (default = '+')] The placement of the arrays relative to each other. Keeping in mind that the origin of an array's index is in the upper-left corner, *'+'* indicates that the second array will be placed at higher index relative to the first array. Essentially

- if how == 'vert'
  - *'+'* -> *a* is above (higher index) *b*
  - *'-'* -> *a* is below (lower index) *b*
- if how == 'horiz'
  - *'+'* -> *a* is to the left of *b*
  - *'-'* -> *a* is to the right of *b*

See the examples and :func:~'pygridtools.misc.padded_stack' for more info.

shift [int, optional (default = 0)] The number of indices the second array should be shifted in axis other than the one being merged. In other words, vertically stacked arrays can be shifted horizontally, and horizontally stacked arrays can be shifted vertically.

Returns

merged [numpy.ndarrays] The unified nodes coordinates.

`pygridtools.core.`**`insert`**(*nodes*, *index*, *axis=0*, *n_nodes=1*)

Inserts new rows or columns between existing nodes.

Parameters

nodes [numpy.ndarray] The the array to be inserted.

index [int] The index within the array that will be inserted.

axis [int] Either 0 to insert rows or 1 to insert columns.

n_nodes [int] The number of new nodes to be inserted. In other words, `n_nodes = 1` implies that the given row to columns will be split in half. Similarly, `n_nodes = 2` will divide into thirds, `n_nodes = 3` implies quarters, and so on.

Returns

inserted [numpy.ndarray] The modified node array.

`pygridtools.core.`**`extract`**(*nodes*, *jstart=None*, *istart=None*, *jend=None*, *iend=None*)

Extracts a subset of an array into new array.

Parameters

jstart, jend [int, optional] Start and end of the selection along the j-index

istart, iend [int, optional] Start and end of the selection along the i-index

Returns

subset [array] The extracted subset of a copy of the original array.

Notes

Calling this without any [j|i][startlend] arguments effectively just makes a copy of the array.

class pygridtools.core.**ModelGrid**(*nodes_x*, *nodes_y*, *crs=None*)
Bases: object

Container for a curvilinear-orthogonal grid. Provides convenient access to masking, manipulation, and visualization methods.

Although a good effort attempt is made to be consistent with the terminology, in general *node* and *vertex* are used interchangeably, with the former prefered over the latter. Similarly, *centroids* and *cells* can be interchangeable, although they are different. (Cell = the polygon created by 4 adjacent nodes and centroid = the centroid point of a cell).

Parameters

nodes_x, nodes_y [numpy.ndarray] M-by-N arrays of node (vertex) coordinates for the grid.

crs [dict or str, optional] Output projection parameters as string or in dictionary form.

nodes_x
Array of node x-coordinates.

nodes_y
Array of node y-coordinates.

cells_x
Array of cell centroid x-coordinates

cells_y
Array of cell centroid y-coordinates

shape
Shape of the nodes arrays

cell_shape
Shape of the cells arrays

xn
Shortcut to x-coords of nodes

yn
Shortcut to y-coords of nodes

xc
Shortcut to x-coords of cells/centroids

yc
Shortcut to y-coords of cells/centroids

icells
Number of rows of cells

jcells
Number of columns of cells

inodes
Number of rows of nodes

jnodes
Number of columns of nodes

cell_mask
Boolean mask for the cells

node_mask

crs
Coordinate reference system for GIS data export

domain
The optional domain used to generate the raw grid

extent
The final extent of the model grid (everything outside is masked).

islands
Polygons used to make holes/gaps in the grid

transform(*fxn*, **args*, ***kwargs*)
Apply an attribrary function to the grid nodes.

Parameters

fxn [callable] The function to be applied to the nodes. It should accept a node array as its first argument.

arg, kwargs [optional arguments and keyword arguments] Additional values passed to `fxn` after the node array.

Returns

modelgrid A new *ModelGrid* is returned.

transform_x(*fxn*, **args*, ***kwargs*)

transform_y(*fxn*, **args*, ***kwargs*)

transpose()
Transposes the node arrays of the model grid.

Parameters

None

Returns

modelgrid A new *ModelGrid* is returned.

fliplr()
Reverses the columns of the node arrays of the model grid.

Parameters

None

Returns

modelgrid A new *ModelGrid* is returned.

flipud()
Reverses the rows of the node arrays of the model grid.

Parameters

None

Returns

modelgrid A new *ModelGrid* is returned.

split(*index*, *axis=0*)
Splits a model grid into two separate objects.

Parameters

index [int] The leading edge of where the split should occur.

axis [int, optional] The axis along which `nodes` will be split. Use *axis = 0* to split along rows and *axis = 1* for columns.

Returns

grid1, grid2 [ModelGrids] The two non-overlapping sides of the grid.

Raises

ValueError Trying to split at the edge (i.e., resulting in the original array and an empty array) will raise an error.

insert(*index*, *axis=0*, *n_nodes=1*)
Inserts and linearly interpolates new nodes in an existing grid.

Parameters

nodes [numpy.ndarray] An N x M array of individual node coordinates (i.e., the x-coords or the y-coords only)

index [int] The leading edge of where the split should occur.

axis [int, optional] The axis along which `nodes` will be split. Use *axis = 0* to split along rows and *axis = 1* for columns.

n_nodes [int, optional] The number of *new* rows or columns to be inserted.

Returns

modelgrid A new *ModelGrid* is returned.

extract(*jstart=0*, *istart=0*, *jend=-1*, *iend=-1*)
Extracts a subset of an array into new grid.

Parameters

jstart, jend [int, optional] Start and end of the selection along the j-index

istart, iend [int, optional] Start and end of the selection along the i-index

Returns

subset [grid] The extracted subset of a copy of the original grid.

Notes

Calling this without any [jli][startlend] arguments effectively just makes a copy of the grid.

copy()
Returns a deep copy of the current ModelGrid

merge(*other*, *how='vert'*, *where='+'*, *shift=0*, *min_nodes=1*)
Merge with another grid using pygridtools.misc.padded_stack.

Parameters

other [ModelGrid] The other ModelGrid object.

how [optional string (default = 'vert')] The method through wich the arrays should be stacked. *'Vert'* is analogous to *numpy.vstack*. *'Horiz'* maps to *numpy.hstack*.

where [optional string (default = '+')] The placement of the arrays relative to each other. Keeping in mind that the origin of an array's index is in the upper-left corner, '+' indicates that the second array will be placed at higher index relative to the first array. Essentially:

- if how == 'vert'
  - '+' -> *a* is above (higher index) *b*
  - '-' -> *a* is below (lower index) *b*
- if how == 'horiz'
  - '+' -> *a* is to the left of *b*
  - '-' -> *a* is to the right of *b*

See the examples and pygridtools.misc.padded_stack for more info.

shift [int (default = 0)] The number of indices the second array should be shifted in axis other than the one being merged. In other words, vertically stacked arrays can be shifted horizontally, and horizontally stacked arrays can be shifted vertically.

min_nodes [int (default = 1)] Minimum number of masked nodes required to mask a cell.

Returns

modelgrid A new *ModelGrid* is returned.

See also:

`pygridtools.padded_stack`

Examples

```
import matplotlib.pyplot as plt
import pandas
import pygridtools
domain1 = pandas.DataFrame({
    'x': [2, 5, 5, 2],
    'y': [6, 6, 4, 4],
    'beta': [1, 1, 1, 1]
})
domain2 = pandas.DataFrame({
    'x': [6, 11, 11, 5],
    'y': [5, 5, 3, 3],
    'beta': [1, 1, 1, 1]
})
grid1 = pygridtools.make_grid(domain=domain1, nx=6, ny=5, rawgrid=False)
grid2 = pygridtools.make_grid(domain=domain2, nx=8, ny=7, rawgrid=False)
merged = grid1.merge(grid2, how='horiz')
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(6, 6))
grid1.plot_cells(ax=ax1, cell_kws=dict(cmap='Blues'))
grid2.plot_cells(ax=ax1, cell_kws=dict(cmap='Greens'))
merged.plot_cells(ax=ax2, cell_kws=dict(cmap='BuPu'))
plt.show()
```

update_cell_mask(*mask=None*, *merge_existing=True*)
Regenerate the cell mask based on either the NaN cells or a user-provided mask. This is usefull after splitting, merging, or anything other transformation.

Parameters

mask [numpy.ndarray of bools, optional] The custom make to apply. If ommited, the mask will be determined by the missing values in the cells arrays.

merge_existing [bool (default is True)] If True, the new mask is bitwise OR'd with the existing mask.

Returns

masked [ModelGrid] A new *ModelGrid* wit the final mask to be applied to the cells.

mask_nodes(*inside=None*, *outside=None*, *min_nodes=3*, *use_existing=False*, *triangles=False*)
Create mask the ModelGrid based on its nodes with a polygon.

Parameters

inside, outside [GeoDataFrame, optional] GeoDataFrames of Polygons or MultiPolygons inside or outside of which nodes will be masked, respectively.

min_nodes [int (default = 3)] Only used when `use_centroids` is False. This is the minimum number of nodes inside the polygon required to mark the cell as "inside". Must be greater than 0, but no more than 4.

use_existing [bool (default = True)] When True, the newly computed mask is combined (via a bit-wise *or* operation) with the existing `cell_mask` attribute of the ModelGrid.

Returns

masked [ModelGrid] A new *ModelGrid* with the final mask to be applied to the cells.

mask_centroids(*inside=None*, *outside=None*, *use_existing=True*)
Create mask for the cells of the ModelGrid with a polygon.

Parameters

inside, outside [GeoDataFrame, optional] GeoDataFrames of Polygons or MultiPolygons inside or outside of which nodes will be masked, respectively.

use_existing [bool (default = True)] When True, the newly computed mask is combined (via a bit-wise *or* operation) with the existing `cell_mask` attribute of the ModelGrid.

Returns

masked [ModelGrid] A new *ModelGrid* wit the final mask to be applied to the cells.

mask_cells_with_polygon(***kwds*)
mask_cells_with_polygon is deprecated! use mask_nodes or mask_centroids

plot_cells(*engine='mpl'*, *ax=None*, *usemask=True*, *showisland=True*, *cell_kws=None*, *domain_kws=None*, *extent_kws=None*, *island_kws=None*)
Creates a figure of the cells, boundary, domain, and islands.

Parameters

engine [str] The plotting engine to be used. Right now, only *'mpl'* has been implemented. Interactive figures via *'bokeh'* are planned.

ax [matplotlib.Axes, optional] The axes onto which the data will be drawn. If not provided, a new one will be created. Applies only to the *mpl* engine.

usemask [bool, optional] Whether or not cells should have the ModelGrid's mask applied to them.

cell_kws, domain_kws, extent_kws, island_kws [dict] Dictionaries of plotting options for each element of the figure.

`cell_kws` and `island_kws` are fed to `Polygon()`. All others are sent to `plot()`.

to_point_geodataframe(*which='nodes'*, *usemask=True*, *elev=None*)

to_polygon_geodataframe(*usemask=True*, *elev=None*)

to_dataframe(*usemask=False*, *which='nodes'*)

Converts a grid to a wide dataframe of coordinates.

Parameters

usemask [bool, optional] Toggles the ommission of masked values (as determined by *cell_mask()*.

which [str, optional ('nodes')] This can be "nodes" (default) or "cells". Specifies which coordinates should be used.

Returns

pandas.DataFrame

to_coord_pairs(*usemask=False*, *which='nodes'*)

Converts a grid to a long array of coordinates pairs.

Parameters

usemask [bool, optional] Toggles the ommission of masked values (as determined by *cell_mask()*.

which [str, optional ('nodes')] This can be "nodes" (default) or "cells". Specifies which coordinates should be used.

Returns

numpy.ndarray

to_gefdc(*directory*)

classmethod from_dataframe(*df*, *icol='ii'*, *jcol='jj'*, *xcol='easting'*, *ycol='northing'*, *crs=None*)

Build a ModelGrid from a DataFrame of I/J indexes and x/y columns.

Parameters

df [pandas.DataFrame] Must have a MultiIndex of I/J cell index values.

xcol, ycol [str, optional] The names of the columns for the x and y coordinates.

icol [str, optional] The index level specifying the I-index of the grid.

crs [dict or str, optional] Output projection parameters as string or in dictionary form.

Returns

ModelGrid

classmethod from_gis(*gisfile*, *icol='ii'*, *jcol='jj'*)

Build a ModelGrid from a GIS file of *nodes*.

Parameters

gisfile [str] The path to the GIS file of the grid *nodes*.

icol, jcol [str, optional] The names of the columns in the *gisfile* containing the I/J index of the nodes.

Returns

ModelGrid

classmethod from_Gridgen(*gridgen*, *crs=None*)

Build a ModelGrid from a `Gridgen` object.

Parameters

gridgen [pygridgen.Gridgen]

crs [dict or str, optional] Output projection parameters as string or in dictionary form.

Returns

ModelGrid

pygridtools.core.**make_grid**(*ny*, *nx*, *domain*, *betacol='beta'*, *crs=None*, *rawgrid=True*, ***gg_params*)

Generate a `Gridgen` or *ModelGrid* from scratch. This can take a large number of parameters passed directly to the `Gridgen` constructor. See the *Other Parameters* section.

Parameters

ny, nx [int] The number of rows and columns that will make up the grid's *nodes*. Note the final grid *cells* will be (ny-1) by (nx-1).

domain [GeoDataFrame] Defines the boundary of the model area. Needs to have a column of Point geometries and a a column of beta (turning point) values.

betacol [str] Label of the column in *domain* that contains the beta (i.e., turning point) values of domain. This sum of this column must be 4.

crs [dict or str, optional] Output projection parameters as string or in dictionary form.

rawgrid [bool (default = True)] When True, returns a pygridgen.Gridgen object. Otherwise, a pygridtools.ModelGrid object is returned.

Returns

grid [pygridgen.Gridgen or ModelGrid]

Other Parameters

ul_idx [optional int (default = 0)] The index of the what should be considered the upper left corner of the grid boundary in the *xbry*, *ybry*, and *beta* inputs. This is actually more arbitrary than it sounds. Put it some place convenient for you, and the algorthim will conceptually rotate the boundary to place this point in the upper left corner. Keep that in mind when specifying the shape of the grid.

focus [optional pygridgen.Focus instance or None (default)] A focus object to tighten/loosen the grid in certain sections.

proj [option pyproj projection or None (default)] A pyproj projection to be used to convert lat/lon coordinates to a projected (Cartesian) coordinate system (e.g., UTM, state plane).

nnodes [optional int (default = 14)] The number of nodes used in grid generation. This affects the precision and computation time. A rule of thumb is that this should be equal to or slightly larger than -log10(precision).

precision [optional float (default = 1.0e-12)] The precision with which the grid is generated. The default value is good for lat/lon coordinate (i.e., smaller magnitudes of boundary coordinates). You can relax this to e.g., 1e-3 when working in state plane or UTM grids and you'll typically get better performance.

nppe [optional int (default = 3)] The number of points per internal edge. Lower values will coarsen the image.

newton [optional bool (default = True)] Toggles the use of Gauss-Newton solver with Broyden update to determine the sigma values of the grid domains. If False simple iterations will be used instead.

thin [optional bool (default = True)] Toggle to True when the (some portion of) the grid is generally narrow in one dimension compared to another.

checksimplepoly [optional bool (default = True)] Toggles a check to confirm that the boundary inputs form a valid geometry.

verbose [optional bool (default = True)] Toggles the printing of console statements to track the progress of the grid generation.

See also:

`pygridgen.Gridgen`, `pygridtools.ModelGrid`

Notes

If your boundary has a lot of points, this really can take quite some time.

5.2 The *iotools* API

`pygridtools.iotools.`**`read_boundary`**(*gisfile*, *betacol='beta'*, *reachcol=None*, *sortcol=None*, *upperleftcol=None*, *filterfxn=None*)

Loads boundary points from a GIS File.

Parameters

gisfile [string] Path to the GIS file containaing boundary points. Expected schema of the file...

- order: numeric sort order of the points
- beta: the 'beta' parameter used in grid generation to define turning points

betacol [string (default='beta')] Column in the attribute table specifying the beta parameter's value at each point.

sortcol [optional string or None (default)] Column in the attribute table specifying the sort order of the points.

reachcol [optional string or None (default)] Column in the attribute table specifying the names of the reaches of the river/esturary system.

upperleftcol [optional string or None (default)] Column in the attribute table toggling if the a point should be consider the upper-left corner of the system. Only one row of this column should evaluare to True.

filterfxn [function or lambda expression or None (default)] Removed. Use the *query* method of the result.

Returns

gdf [geopandas.GeoDataFrame] A GeoDataFrame of the boundary points with the following columns:

- x (easting)
- y (northing)
- beta (turning parameter)
- order (for sorting)
- reach
- upperleft

pygridtools.iotools.**read_polygons**(*gisfile*, *filterfxn=None*, *squeeze=True*, *as_gdf=False*)

Load polygons (e.g., water bodies, islands) from a GIS file.

Parameters

gisfile [string] Path to the gisfile containaing boundary points.

filterfxn [function or lambda expression or None (default)] Removed. Use the *as_gdf* and the *query* method of the resulting GeoDataFrame.

squeeze [optional bool (default = True)] Set to True to return an array if only 1 record is present. Otherwise, a list of arrays will be returned.

as_gdf [optional bool (default = False)] Set to True to return a GeoDataFrame instead of arrays.

Returns

boundary [array or list of arrays, or GeoDataFrame]

Notes

Multipart geometries are not supported. If a multipart geometry is present in a record, only the first part will be loaded.

Z-coordinates are also not supported. Only x-y coordinates will be loaded.

pygridtools.iotools.**read_grid**(*gisfile*, *icol='ii'*, *jcol='jj'*, *othercols=None*, *expand=1*, *as_gdf=False*)

pygridtools.iotools.**interactive_grid_shape**(*grid*, *max_n=200*, *plotfxn=None*, ***kwargs*)

Interactive ipywidgets for select the shape of a grid

Parameters

grid [pygridgen.Gridgen] The base grid from which the grids of new shapes (resolutions) will be generated.

max_n [int (default = 200)] The maximum number of possible cells in each dimension.

plotfxn [callable, optional] Function that plots the grid to provide user feedback. The call signature of this function must accept to positional parameters for the x- and y-arrays of node locations, and then accept any remaining keyword arguments. If not provided, *pygridtools.viz.plot_cells* is used.

Returns

newgrid [pygridgen.Gridgen] The reshaped grid

widget [ipywidgets.interactive] Collection of IntSliders for changing the number cells along each axis in the grid.

Examples

```
>>> from pygridgen import grid
>>> from pygridtools import viz, iotools
>>> def make_fake_bathy(shape):
...     j_cells, i_cells = shape
...     y, x = numpy.mgrid[:j_cells, :i_cells]
...     z = (y - (j_cells // 2))** 2 - x
...     return z
>>> def plot_grid(x, y, ax=None):
...     shape = x[1:, 1:].shape
...     bathy = make_fake_bathy(shape)
...     if not ax:
...         fig, ax = pyplot.subplots(figsize=(8, 8))
...     ax.set_aspect('equal')
...     return viz.plot_cells(x, y, ax=ax, cmap='Blues', colors=bathy, lw=0.5, ec=
↪'0.3')
>>> d = numpy.array([
... (13, 16,  1.00), (18, 13,  1.00), (12,  7,  0.50),
... (10, 10, -0.25), ( 5, 10, -0.25), ( 5,  0,  1.00),
... ( 0,  0,  1.00), ( 0, 15,  0.50), ( 8, 15, -0.25),
... (11, 13, -0.25)])
>>> g = grid.Gridgen(d[:, 0], d[:, 1], d[:, 2], (75, 75), ul_idx=1, focus=None)
>>> new_grid, widget = iotools.interactive_grid_shape(g, plotfxn=plot_grid)
```

5.3 The *misc* API

pygridtools.misc.**make_poly_coords**(*xarr*, *yarr*, *zpnt=None*, *triangles=False*)

Makes an array for coordinates suitable for building quadrilateral geometries in shapfiles via fiona.

Parameters

xarr, yarr [numpy arrays] Arrays (2x2) of x coordinates and y coordinates for each vertex of the quadrilateral.

zpnt [optional float or None (default)] If provided, this elevation value will be assigned to all four vertices.

triangles [optional bool (default = False)] If True, triangles will be returned

Returns

coords [numpy array] An array suitable for feeding into fiona as the geometry of a record.

pygridtools.misc.**make_record**(*ID*, *coords*, *geomtype*, *props*)

Creates a record to be appended to a GIS file via *geopandas*.

Parameters

ID [int] The record ID number

coords [tuple or array-like] The x-y coordinates of the geometry. For Points, just a tuple. An array or list of tuples for LineStrings or Polygons

geomtype [string] A valid GDAL/OGR geometry specification (e.g. LineString, Point, Polygon)

props [dict or collections.OrderedDict] A dict-like object defining the attributes of the record

Returns

record [dict] A nested dictionary suitable for the a *geopandas.GeoDataFrame*,

Notes

This is ignore the mask of a MaskedArray. That might be bad.

pygridtools.misc.**interpolate_bathymetry**(*bathy*, *x_points*, *y_points*, *xcol='x'*, *ycol='y'*, *zcol='z'*)

Interpolates x-y-z point data onto the grid of a Gridgen object. Matplotlib's nearest-neighbor interpolation schema is used to estimate the elevation at the grid centers.

Parameters

bathy [pandas.DataFrame or None] The bathymetry data stored as x-y-z points in a DataFrame.

[x|y]_points [numpy arrays] The x, y locations onto which the bathymetry will be interpolated.

xcol/ycol/zcol [optional strings] Column names for each of the quantities defining the elevation pints. Defaults are "x/y/z".

Returns

gridbathy [pandas.DataFrame] The bathymetry for just the area covering the grid.

pygridtools.misc.**padded_stack**(*a*, *b*, *how='vert'*, *where='+'*, *shift=0*, *padval=nan*)

Merge 2-dimensional numpy arrays with different shapes.

Parameters

a, b [numpy arrays] The arrays to be merged

how [optional string (default = 'vert')] The method through wich the arrays should be stacked. *'Vert'* is analogous to *numpy.vstack*. *'Horiz'* maps to *numpy.hstack*.

where [optional string (default = '+')] The placement of the arrays relative to each other. Keeping in mind that the origin of an array's index is in the upper-left corner, *'+'* indicates that the second array will be placed at higher index relative to the first array. Essentially:

- **if how == 'vert'**
  - '+' -> *a* is above (higher index) *b*
  - '-' -> *a* is below (lower index) *b*
- **if how == 'horiz'**
  - '+' -> *a* is to the left of *b*
  - '-' -> *a* is to the right of *b*

See the examples for more info.

shift [int (default = 0)] The number of indices the second array should be shifted in axis other than the one being merged. In other words, vertically stacked arrays can be shifted horizontally, and horizontally stacked arrays can be shifted vertically.

padval [optional, same type as array (default = numpy.nan)] Value with which the arrays will be padded.

Returns

Stacked [numpy array] The merged and padded array

Examples

```
>>> import pygridtools as pgt
>>> a = numpy.arange(12).reshape(4, 3) * 1.0
>>> b = numpy.arange(8).reshape(2, 4) * -1.0
>>> pgt.padded_stack(a, b, how='vert', where='+', shift=2)
array([[ 0.,  1.,  2.,  -,  -,  -],
       [ 3.,  4.,  5.,  -,  -,  -],
       [ 6.,  7.,  8.,  -,  -,  -],
       [ 9., 10., 11.,  -,  -,  -],
       [  -,  -, -0., -1., -2., -3.],
       [  -,  -, -4., -5., -6., -7.]])
```

```
>>> pgt.padded_stack(a, b, how='h', where='-', shift=-1)
array([[-0., -1., -2., -3.,  -,  -,  -],
       [-4., -5., -6., -7.,  0.,  1.,  2.],
       [  -,  -,  -,  -,  3.,  4.,  5.],
       [  -,  -,  -,  -,  6.,  7.,  8.],
       [  -,  -,  -,  -,  9., 10., 11.]])
```

pygridtools.misc.**padded_sum**(*padded*, *window=1*)

pygridtools.misc.**mask_with_polygon**(*x*, *y*, **polyverts*, *inside=True*)

Mask x-y arrays inside or outside a polygon

Parameters

x, y [array-like] NxM arrays of x- and y-coordinates.

polyverts [sequence of a polygon's vertices] A sequence of x-y pairs for each vertex of the polygon.

inside [bool (default is True)] Toggles masking the inside or outside the polygon

Returns

mask [bool array] The NxM mask that can be applied to `x` and `y`.

pygridtools.misc.**gdf_of_cells**(*X*, *Y*, *mask*, *crs*, *elev=None*, *triangles=False*)

Saves a GIS file of quadrilaterals representing grid cells.

Parameters

X, Y [numpy (masked) arrays, same dimensions] Attributes of the gridgen object representing the x- and y-coords.

mask [numpy array or None] Array describing which cells to mask (exclude) from the output. Shape should be N-1 by M-1, where N and M are the dimensions of *X* and *Y*.

crs [string] A geopandas/proj/fiona-compatible string describing the coordinate reference system of the x/y values.

elev [optional array or None (defauly)] The elevation of the grid cells. Shape should be N-1 by M-1, where N and M are the dimensions of *X* and *Y* (like *mask*).

triangles [optional bool (default = False)] If True, triangles can be included

Returns

geopandas.GeoDataFrame

pygridtools.misc.**gdf_of_points**(*X*, *Y*, *crs*, *elev=None*)

Saves grid-related attributes of a pygridgen.Gridgen object to a GIS file with geomtype = 'Point'.

Parameters

X, Y [numpy (masked) arrays, same dimensions] Attributes of the gridgen object representing the x- and y-coords.

crs [string] A geopandas/proj/fiona-compatible string describing the coordinate reference system of the x/y values.

elev [optional array or None (defauly)] The elevation of the grid cells. Array dimensions must be 1 less than X and Y.

Returns

geopandas.GeoDataFrame

5.4 The *gefdc* API

pygridtools.gefdc.**write_cellinp**(*cell_array*, *outputfile='cell.inp'*, *mode='w'*, *writeheader=True*, *rowlabels=True*, *maxcols=125*, *flip=True*)

Writes the cell.inp input file from an array of cell definitions.

Parameters

cell_array [numpy array] Integer array of the values written to `outfile`.

outputfile [optional string (default = "cell.inp")] Path *and* filename to the output file. Yes, you have to tell it to call the file cell.inp

maxcols [optional int (default = 125)] Number of columns at which cell.inp should be wrapped. `gefdc` requires this to be 125.

flip [optional bool (default = True)] Numpy arrays have their origin in the upper left corner, so in a sense south is up and north is down. This means that arrays need to be flipped before writing to "cell.inp". Unless you are _absolutely_sure_ that your array has been flipped already, leave this parameter as True.

Returns

None

See also:

make_gefdc_cells

pygridtools.gefdc.**write_gefdc_control_file**(*outfile*, *title*, *max_i*, *max_j*, *bathyrows*)

pygridtools.gefdc.**write_gridout_file**(*xcoords*, *ycoords*, *outfile*)

pygridtools.gefdc.**write_gridext_file**(*tidydf*, *outfile*, *icol='ii'*, *jcol='jj'*, *xcol='easting'*, *ycol='northing'*)

pygridtools.gefdc.**convert_gridext_to_gis**(*inputfile*, *outputfile*, *crs=None*, *river='na'*, *reach=0*)

Converts gridext.inp from the rtools to a GIS file with *geomtype = 'Point'*.

Parameters

inputfile [string] Path and filename of the gridext.inp file

outputfile [string] Path and filename of the destination GIS file

crs [string, optional] A geopandas/proj/fiona-compatible string describing the coordinate reference system of the x/y values.

river [optional string (default = None)] The river to be listed in the output file's attributes.

reach [optional int (default = 0)] The reach of the river to be listed in the output file's attributes.

Returns

geopandas.GeoDataFrame

pygridtools.gefdc.**make_gefdc_cells**(*node_mask*, *cell_mask=None*, *triangles=False*)

Take an array defining the nodes as wet (1) or dry (0) create the array of cell values needed for GEFDC.

Parameters

node_mask [numpy bool array (N x M)] Bool array specifying if a *node* is present in the raw (unmasked) grid.

cell_mask [optional numpy bool array (N-1 x M-1) or None (default)] Bool array specifying if a cell should be masked (e.g. due to being an island or something like that).

triangles [optional bool (default = False)] Currently not implemented. Will eventually enable the writting of triangular cells when True.

Returns

cell_array [numpy array] Integer array of the values written to `outfile`.

class pygridtools.gefdc.**GEFDCWriter**(*mg*, *directory*)

Bases: `object`

Convenience class to write the GEFDC files for a ModelGrid

Parameters

mg [pygridtools.ModelGrid]

directory [str or Path] Where all of the files will be saved

control_file(*filename='gefdc.inp'*, *bathyrows=0*, *title=None*)

Generates the GEFDC control (gefdc.inp) file for the EFDC grid preprocessor.

Parameters

filename [str, optional] The name of the output file.

bathyrows [int, optional] The number of rows in the grid's bathymetry data file.

title [str, optional] The title of the grid as portrayed in `filename`.

Returns

gefdc [str] The text of the output file.

cell_file(*filename='cell.inp'*, *triangles=False*, *maxcols=125*)

Generates the cell definition/ASCII-art file for GEFDC.

Parameters

filename [str, optional] The name of the output file.

triangles [bool, optional] Toggles the inclusion of triangular cells.

maxcols [int, optional] The maximum number of columns to write to each row. Cells beyond this number will be written in separate section at the bottom of the file.

Returns

cells [str] The text of the output file.

gridout_file (*filename='grid.out'*)

Writes to the nodes as coordinate pairs for GEFDC.

Parameters

filename [str, optional] The name of the output file.

Returns

df [pandas.DataFrame] The dataframe of node coordinate pairs.

gridext_file (*filename='gridext.inp'*, *shift=2*)

Writes to the nodes and I/J cell index as to a file for GEFDC.

Parameters

filename [str, optional] The name of the output file.

shift [int, optional] The shift that should be applied to the I/J index. The default value to 2 means that the first cell is at (2, 2) instead of (0, 0).

Returns

df [pandas.DataFrame] The dataframe of coordinates and I/J index.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

U

W

X

Y