gettingstarted_excercise

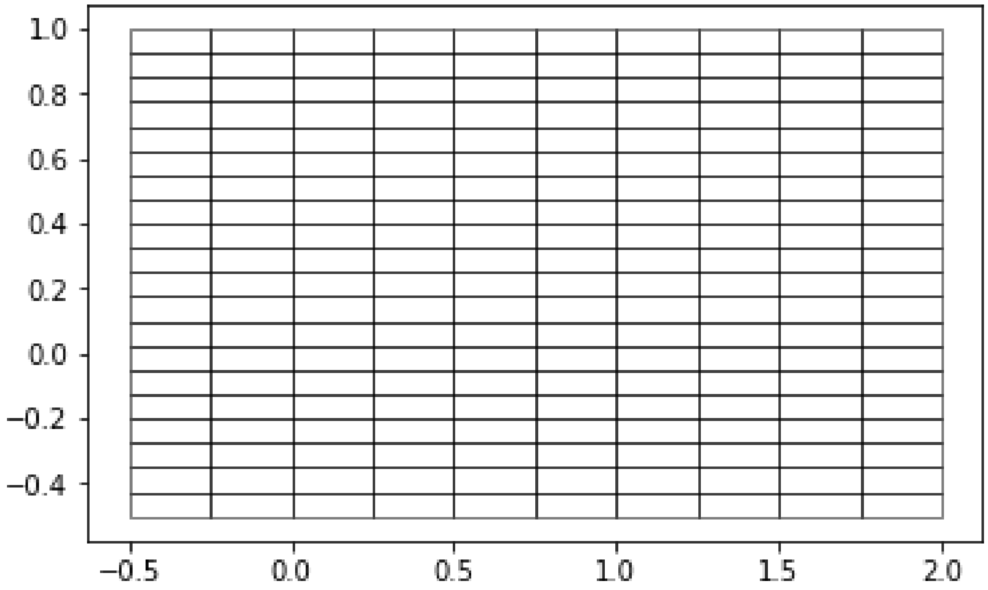
April 3, 2022

1 Grid Construction

The simplest approach to construct a grid is to use the `structuredGrid` functions which takes coordinates of the lower left and upper right corner of the domain and the subdivision to use in each coordinate direction.

Setup a Cartesian grid for $[-\frac{1}{2}, -\frac{1}{2}] \times [2, 1]$ with 10 cells in the x direction and 20 in the y direction:

```
2022-04-03 16:03:49,560 - dune.generator.cmakebuilder - INFO - Compiling
HierarchicalGrid (rebuilding)
```



To construct more complicated grids, we can use Python dictionaries which stores key/value pairs in the form

`` `{key1:value1, key2:value2,...}`. ``

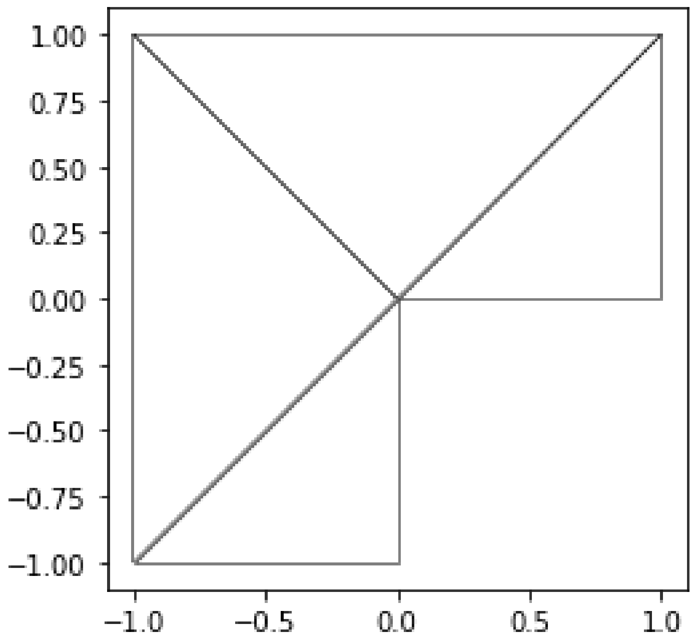
For example construct some triangular grid using 'ALUGrid'

with four triangles and corners $(0, 0), (1, 0), (1, 1), (-1, 1), (-1, -1), (0, -1)$:

```
Created parallel ALUGrid<2,2,simplex,conforming> from input stream.

WARNING (ignored): Could not open file 'alugrid.cfg', using default values 0 <
[balance] < 1.2, partitioning method 'ALUGRID_SpaceFillingCurve(9)'.

You are using DUNE-ALUGrid, please don't forget to cite the paper:
Alkaemper, Dedner, Kloefkorn, Nolte. The DUNE-ALUGrid Module, 2016.
```



1.1 Using the grid:

The most common task is to iterate over the elements (codimension zero entities) of the grid. This can be done with a simple for loop

```
for element in view.elements:
```

The element provides all its geometric information through the `geometry` property. The returned object provides all information concerning the mapping from reference into physical space, i.e., the center of the element its volume, the reference mapping, or the coordinats of the corners using `element.geometry.corners` which returns a vector of the corner points (in fact it returns a list of `Dune::FiledVector` instances.

Print the volume and the center for all elements. There are two ways to obtain the center - try both:

```
0.5 (-0.333333, -0.666667) (-0.333333, -0.666667)
1.0 (-0.666667, 0.000000) (-0.666667, 0.000000)
```

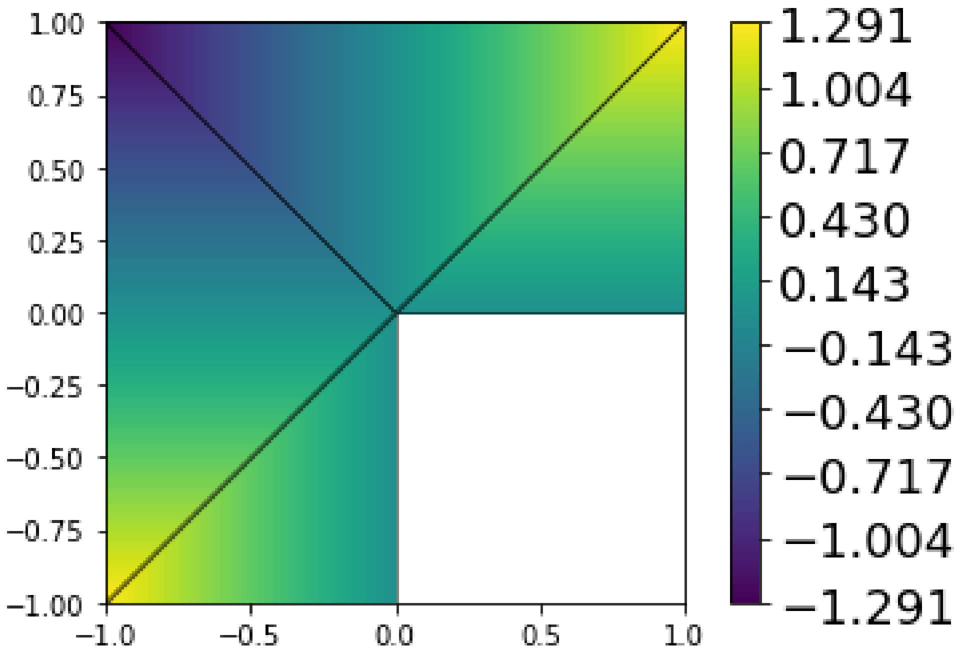
```
0.5 (0.666667, 0.333333) (0.666667, 0.333333)
1.0 (0.000000, 0.666667) (0.000000, 0.666667)
```

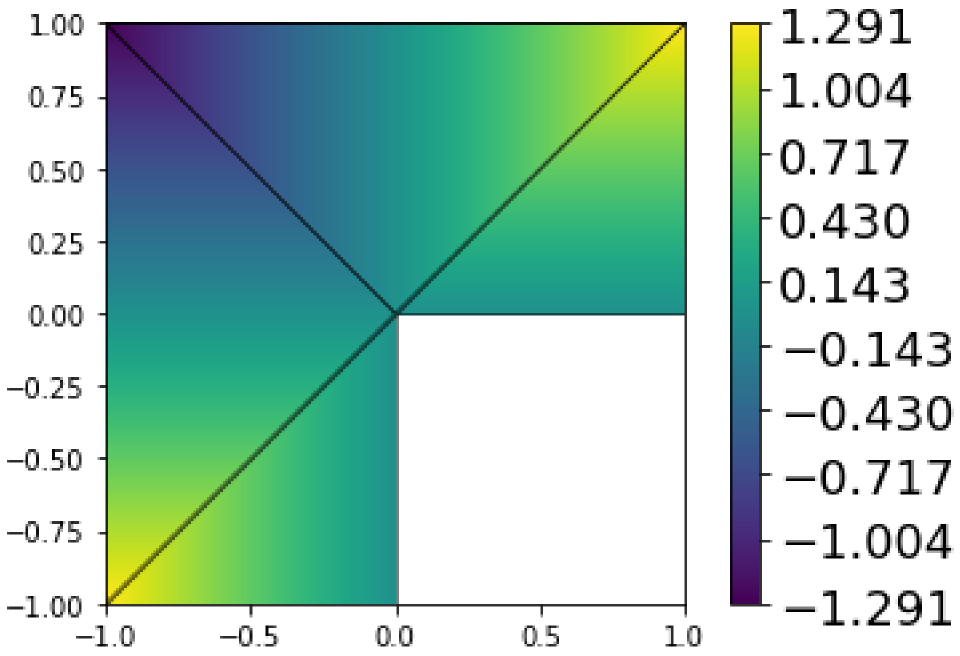
1.2 Grid Functions

It's also quite easy to construct functions defined over the grid, i.e., grid-functions. These are functions that can be evaluated given an element and a local coordinate, i.e., a coordinate in the reference element.

Grid functions are usual Python functions either of the form `def f(globalx)` or `def f(element,localx)` using a Python decorator - the decorator provides some extra functionality like a `plot` function.

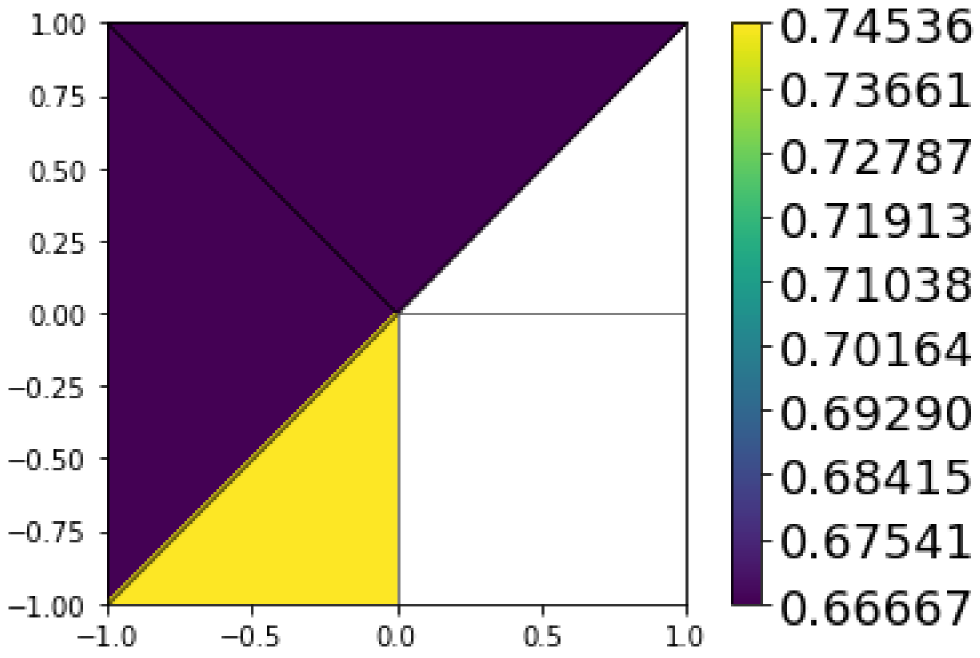
Use both versions to write a grid function for $|x|\ \sin(20x_1x_2)$ and plot the resulting grid function:





Write a piecewise constant grid function that on each element returns the abosolute value of the center of the element.

Note: the return value of `geometry.center` provides an attribute `two_norm`:

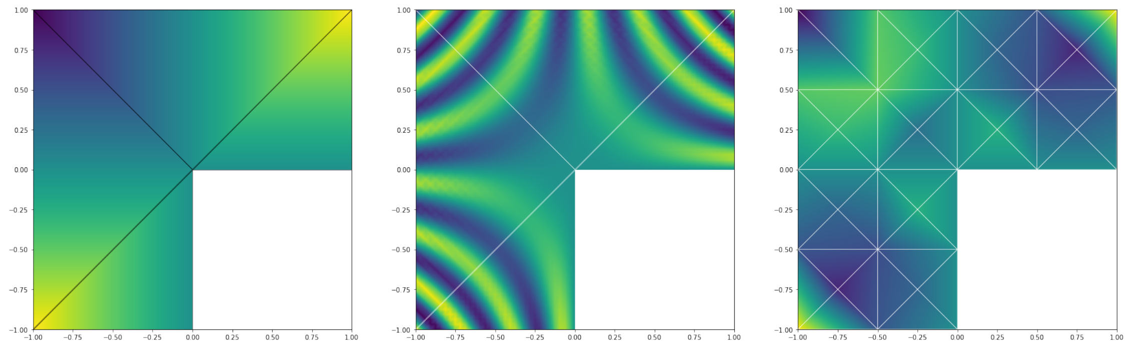


Independent of the signature of the function used, one can evaluate each grid function given an element and a local coordinate.

Iterate over the grid and print the value of all your your grid functions in the center of the elements.

```
-0.7187595417731203 -0.7187595417731203 0.7453559924999299
-9.868649107779172e-16 -9.868649107779172e-16 0.6666666666666667
-0.718759541773121 -0.718759541773121 0.74535599249993
9.868649107779172e-16 9.868649107779172e-16 0.6666666666666667
```

Refine the grid a view times (three times for example) and plot all functions again:



Finally we can write vtk files which can be viewed with some third party software e.g. `paraview` -

```
view.writeVTK("sin", pointdata=[f1,f2,center])
```

1.3 Attaching data to the grid

For this we use a `mapper` g_T provided by the grid. It is constructed by passing in a `layout`, i.e., how many degrees of freedom are needed per subentity (in fact per `geometry` type…)

Let's attach two degree of freedom to each vertex and one degree of freedom to each edge and print the number of dofs for a Cartesian grid with 3x2 cells:

```
Created parallel ALUGrid<2,2,simplex,conforming> from input stream.

number of degrees of freedom: 47
```

```
GridParameterBlock: Parameter 'refinementedge' not specified, defaulting to
'ARBITRARY'.
WARNING (ignored): Could not open file 'alugrid.cfg', using default values 0 <
[balance] < 1.2, partitioning method 'ALUGRID_SpaceFillingCurve(9)'.
```

The easiest way to access all indices attach to an element, i.e., the vector $\big(\mu_T(i)\big)_i$, # is to call the method `indices=mapper(element)`.

The vector of indices contains first the indices for the vertices then come the edges,…, down (or up) to the element's global indices.

The order for each set of subentities depends on their numbering in the Dune reference element.

Iterate over the grid and print the indices attached to each element: Draw a sketch of the grid and try to see if you understand the results...

```
all indices on element: [23 24 25 26 27 28  0  1  2]
all indices on element: [29 30 31 32 25 26  3  4  5]
all indices on element: [27 28 33 34 23 24  6  1  7]
all indices on element: [25 26 23 24 29 30  0  4  8]
all indices on element: [35 36 37 38 31 32  9 10 11]
all indices on element: [31 32 29 30 35 36  3 10 12]
all indices on element: [39 40 23 24 33 34 13 14  7]
all indices on element: [33 34 41 42 39 40 15 14 16]
all indices on element: [43 44 29 30 23 24 17 18  8]
all indices on element: [23 24 39 40 43 44 13 18 19]
all indices on element: [45 46 35 36 29 30 20 21 12]
all indices on element: [29 30 43 44 45 46 17 21 22]
```

Now to some 'real' example: we compute the linear interpolation of a grid function u over a triangular grid.

We need to attach one degree of freedom to each vertex p which corresponds to the value of the function at p. We store the degrees of freedom in a numpy array.

On each triangle with vertices p_0, p_1, p_2 the discrete function is now given by the unique linear function taking the values $u(p_0), u(p_1), u(p_2)$ at the three vertices. This function is

$$u_h(e, \hat{x}) = u(p_0)(1 - \hat{x}_0 - \hat{x}_1) + u(p_1)\hat{x}_0 + u(p_2)\hat{x}_1$$

where $\hat{x}_0, \hat{x}_1$ is the coordinate in the reference element.

Write a function `linear interpolation` that takes a grid function `u` and a grid view and returns the mapper and vector of degrees of freedom (dofs), i.e., the numpy array storing the values of `u` at the vertices.

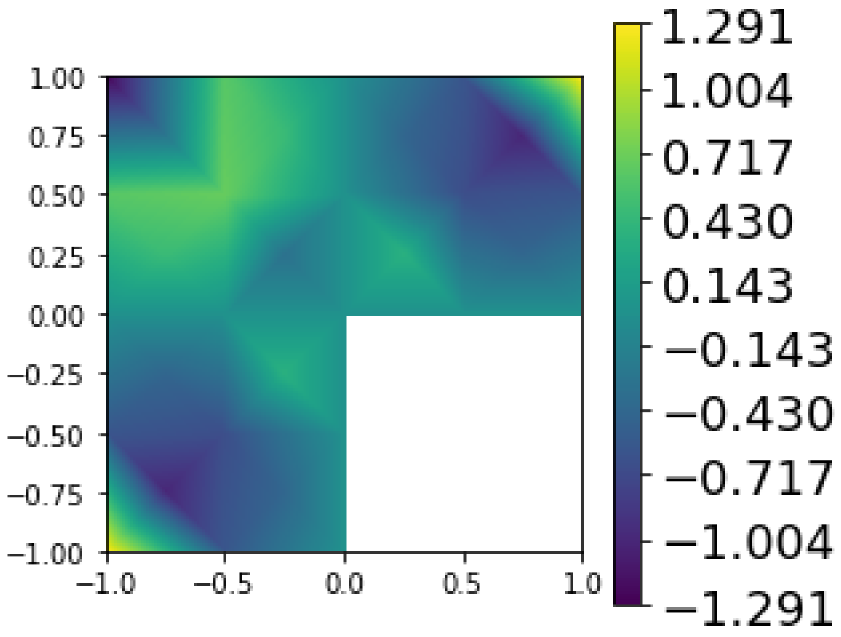
Note: note that a grid function has an attribute `view`.

Then write a grid function `u_h` that returns the linear interpolation for the given dof vector and mapper.

Use the unstructured grid you setup before and refine it 3 time globally.

```
WARNING (ignored): Could not open file 'alugrid.cfg', using default values 0 <
[balance] < 1.2, partitioning method 'ALUGRID_SpaceFillingCurve(9)'.


Created parallel ALUGrid<2,2,simplex,conforming> from input stream.
```



Let’s compute the maximum interpolation error by comparing the linear interpolation `u_h` at the center of each element with the value of `u` and printing the maximum value:

```
number of elements: 46 maximum error at element center: 1.500215648026925
```