2DMultigrid_Exam

July 17, 2020

1 2D Multigrid for Darcy Flow

[Total : 34 points]

In the exercises, we have explored some steps towards multigrid, mostly for the Poisson equation and simple discretizations. Furthermore, we have only discussed one-dimensional problems. The problem is, of course, that both one-dimensional problems and the Poisson equation are relatively tame from a numerical perspective. You don't see the advantages of multigrid nn these scenarios,

In this worksheet, we are going to look at the Darcy flow equation, discretized with a simple finite elements scheme on a hierarchical quad-tree based grid. All concepts are exactly the same as you've seen in the course, but applied in a more difficult setting.

[1]:
```
%matplotlib inline
import numpy as np

import time
import scipy.special as special
import scipy.sparse as sp
import scipy.sparse.linalg as splinalg

import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.cm as cm
from matplotlib import patches

import sys
sys.path.append("../src/")

import quadtree
import plotter
import fem
```

1.1 The PDE: Darcy Flow

We are solving the steady state Darcy flow equation. This equation states the law of fluid flow through a porous medium, e.g. through sand.

We have the steady-state fluid pressure u and the spatially varying permeability $K(x, y)$. The PDE is basically a variable-coefficient version of the Poission equation and looks like:

$$-\nabla \cdot (K(x,y)\nabla u) = f, \qquad \forall \mathbf{x} \in \Omega$$

with boundary conditions

$$u(x,y) = 0 \qquad \forall x \in \Gamma_D \tag{1}$$
$$-K(x)\nabla u \cdot n = 0 \qquad \forall x \in \Gamma - \Gamma_D \tag{2}$$

where Γ is the boundary and Γ_D is the Dirichlet part of the boundary.

The Dirichlet boundary is on the top of the domain, all other boundary conditions are set to Neumann.

After multiplying by a testfunction $\phi(x,y)$ and integrating over the domain, we get the weak form:

$$\int_\Omega \nabla\phi(x,y) \cdot (K(x,y)\nabla u(x,y)\mathrm{d}\Omega = \int_\Omega \phi(x,y) f(x,y)\mathrm{d}\Omega$$

The boundary terms are zero due to the choice of boundary conditions. In our case we use a constant right-hand side of $f(x,y) = 1$.

2 The grid

We define a quadtree-based grid, as this allows us to perform hierarchical algorithms with ease. Briefly, we decompose the domain into squares.

3 Discretization

For the discretization we use a simple linear continuous finite elements approach.

We choose a nodal linear Lagrange basis, with points $x_1 = -1, x_2 = 1$. This was discussed in the lecture as a piecewise-linear hat basis. The 1D-basis functions are then

$$l_1(x) = -\frac{x-1}{2} \tag{3}$$
$$l_2(x) = +\frac{x+1}{2} \tag{4}$$
$$\tag{5}$$

We combine them by

$$L(x) = a_1 l_1(x) + a_2 l_2(x)$$

where a_1, a_2 are coefficients. In 2D, we have the points $(-1,-1), (-1,1), (1,-1), (1,1)$ and the solution is expressed as

$$u(x,y) \approx a_1 l_1(x) l_2(y) + a_2 l_1(x) l_2(y) + a_3 l_1(x) l_2(y) + a_4 l_1(x) l_2(y)$$

Note that each vertex has a coefficient (the value of $u(x,y)$ at that point) and a corresponding basis function. We use this basis to represent both the test functions $\phi(x,y)$ and the discrete solution $u(x,y)$.

Furthermore, be careful that all basis functions are defined on the reference cell $[-1,1]^2$ but all grid coordinates are defined in terms of global coordinates.

Finally, we end up with a stiffness matrix A_{ij} and a right-hand side f_i, where i and j are indices of vertices. It is important to note here that we remove rows and columns where either i or j corresponds to a vertex where the Dirichlet boundary condition is enfored. We do this to ensure that the remaining system is symmetric.

This is all you need to know about the discretization to finish this notebook. If you are interested, you can find details in the file *fem.py*. This might also supply you with hints about details regarding the grid, basis, etc.

[2]:
```
# Here we build up the geometry.
# The code is included in the notebook because it makes it easier to see
# how the vertices get numbered.
class Geometry:
    def __init__(self, level):
        center = np.array([0.5, 0.5])
        size = 1.0
        self.grid = quadtree.Quadtree(center=center, size=size)
        self.grid.split_to_level(level)

        self.vertices_coords_to_idx, \
        self.number_of_vertices_per_level, \
        self.boundary_vertices = self.grid.get_vertices()
        self.vertices_idx_to_coords = {v: k for k, v in self.
↪vertices_coords_to_idx.items()}
        self.grid.set_all_cell_vertices(self.vertices_coords_to_idx)
        self.grid.set_all_cell_indices()

        self.dirichlet_vertices = set()
        for v in self.boundary_vertices:
            eps = 1e-8
            if self.vertices_idx_to_coords[v][1] >= (1 - eps):
                self.dirichlet_vertices.add(v)

        dirichlet_vertices_array = np.array([*self.dirichlet_vertices])
        # We do not store the data for Dirichlet values

        self.data_size_per_level = []
        for level in range(self.grid.get_max_level()+1):
            n_vert = self.number_of_vertices_per_level[level]
```

```
            n_dirichlet_vert = len(
                dirichlet_vertices_array[
                    dirichlet_vertices_array < n_vert])
            self.data_size_per_level.append(n_vert - n_dirichlet_vert)


        self.vertex_idx_to_data_idx = np.zeros(self.
↪number_of_vertices_per_level[-1], dtype=np.int32)
        self.data_idx_to_vertex_idx = np.zeros(self.data_size_per_level[-1],␣
↪dtype=np.int32)
        cur_data_idx = 0
        for i in range(self.number_of_vertices_per_level[-1]):
            if i in self.dirichlet_vertices:
                self.vertex_idx_to_data_idx[i] = -1
            else:
                self.vertex_idx_to_data_idx[i] = cur_data_idx
                self.data_idx_to_vertex_idx[cur_data_idx] = i
                cur_data_idx += 1


geometry = Geometry(level=4)
geometry.number_of_vertices_per_level[0]
```

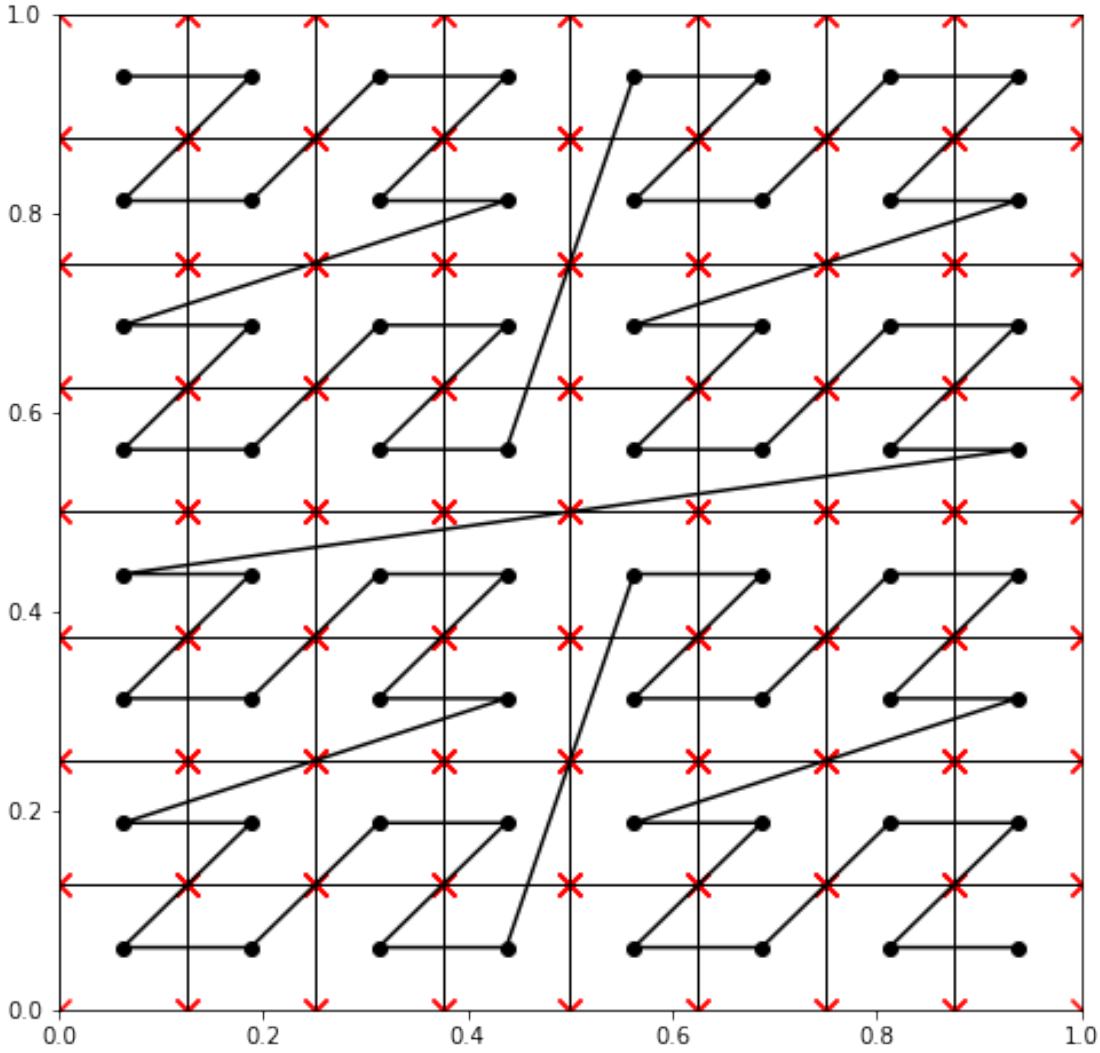
```
Level 0 has     4 vertices
Level 1 has     5 vertices
Level 2 has     16 vertices
Level 3 has     56 vertices
Level 4 has     208 vertices
```

[2]: 4

The figure below shows the grid for level 3. The red crosses show the vertices (at which the solution is defined), the black circle shows the cell center which is connected by the line. This line shows the order of cells (they are in Z-order), starting from the top-left.

[3]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(8,8))
ax.set_xlim(geometry.grid.offset[0], geometry.grid.size)
ax.set_ylim(geometry.grid.offset[1], geometry.grid.size)
plotter.plot_grid(fig,
                  ax,
                  geometry.grid,
                  level=3,
                  c='black',
                  vertices=geometry.vertices_idx_to_coords,
                  plot_curve=True,
                  plot_center=True,
```

```
                    plot_vertices=True)
```



3.1 Solving the system for a single level

Below we show how to setup a discretization, geometry and stiffness matrix and rhs. We also show how to compute a solution with a direct solver.

[4]:
```
# Use a simple material
def eval_k(x,y):
    return x + y

level = 4
geometry = Geometry(level=4)
discretization = fem.Discretization(geometry,level=level,eval_k=eval_k)
```

```
stiffness = discretization.setup_stiffness()
rhs = discretization.setup_rhs()
sol = splinalg.linsolve.spsolve(stiffness, rhs)

# System is solved correctly to double precision:
np.linalg.norm(stiffness @ sol - rhs)
```

```
Level 0 has     4 vertices
Level 1 has     5 vertices
Level 2 has     16 vertices
Level 3 has     56 vertices
Level 4 has     208 vertices
```

[4]: `8.390001524265875e-15`

3.2 Plotting the solution

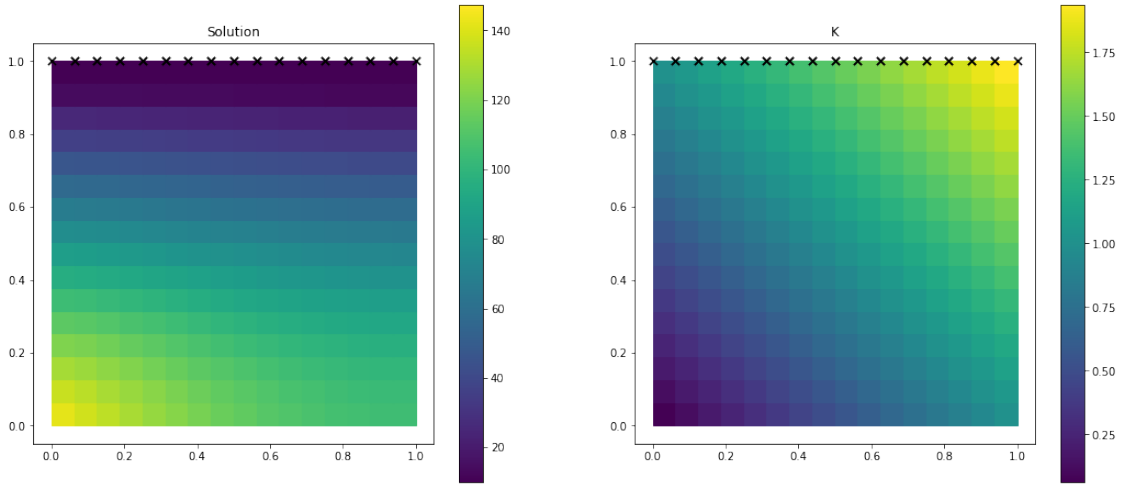
You can visualize the solution with *plot_solution* as below. It also kept in the notebook as a further example of a grid traversal, for the evaluation of the basis functions and for the handling of Dirichlet boundary conditions.

The left plot shows the solution, the right one shows the material parameter. Dirichlet vertices are marked with black x.

[5]:
```
def plot_solution(geometry, discretization, sol):
    # First find minimum and maximum material parameter
    min_k, max_k = float('inf'), float('-inf')
    for cell in geometry.grid.dfs(only_level=level):
        k = discretization.eval_k(cell.center[0], cell.center[1])
        min_k = min(k, min_k)
        max_k = max(k, max_k)

    # Set up color scales for the solution and the material k.
    norm_solution = mpl.colors.Normalize(vmin=sol.min(), vmax=sol.max())
    norm_k = mpl.colors.Normalize(vmin=min_k, vmax=max_k)
    cmap = cm.viridis
    color_mapper_solution = cm.ScalarMappable(norm=norm_solution, cmap=cmap)
    color_mapper_k = cm.ScalarMappable(norm=norm_k, cmap=cmap)

    fig, axs = plt.subplots(1,2, figsize=(18,8))

    # Lists to store Dirichlet vertices.
    dirichlet_xs = []
    dirichlet_ys = []

    for cell in geometry.grid.dfs(only_level=level):
        value = 0.0
        # Evaluate all basis functions at (0.5, 0.5)
```

```
        for i, vertex in enumerate(cell.vertices):
            # Ignore dirichlet boundary conditions.
            # Their value is not stored and is always zero.
            if vertex not in geometry.dirichlet_vertices:
                # Here we get the index of the data correspond to the vertex.
                data_idx = geometry.vertex_idx_to_data_idx[vertex]
                value += sol[data_idx] * fem.lagrange_2d(discretization.
↪nodes_x, i, (0.5,0.5))

        # Plot the rectangles that build up the grid.
        rect = patches.Rectangle(
            cell.offset,
            cell.size,
            cell.size,
            fill=True,
            color=color_mapper_solution.to_rgba(value))
        axs[0].add_patch(rect)

        # Plot material
        value = discretization.eval_k(cell.center[0], cell.center[1])
        rect = patches.Rectangle(
            cell.offset,
            cell.size,
            cell.size,
            fill=True,
            color=color_mapper_k.to_rgba(value))
        axs[1].add_patch(rect)

        # Find dirichlet vertices
        for v in cell.vertices:
            if v in geometry.dirichlet_vertices:
                dirichlet_xs.append(geometry.vertices_idx_to_coords[v][0])
                dirichlet_ys.append(geometry.vertices_idx_to_coords[v][1])

    axs[0].set_title("Solution")
    axs[0].axis('square')

    axs[1].set_title("K")
    axs[1].axis('square')

    # Mark Dirichlet vertices with black x.
    for ax in axs:
        ax.scatter(dirichlet_xs, dirichlet_ys, c='black', zorder=10,␣
↪marker='x', s=50)

    fig.colorbar(color_mapper_solution, ax=axs[0])
    fig.colorbar(color_mapper_k, ax=axs[1])
```

```
plot_solution(geometry=geometry, discretization=discretization, sol=sol)
```



4 Multigrid

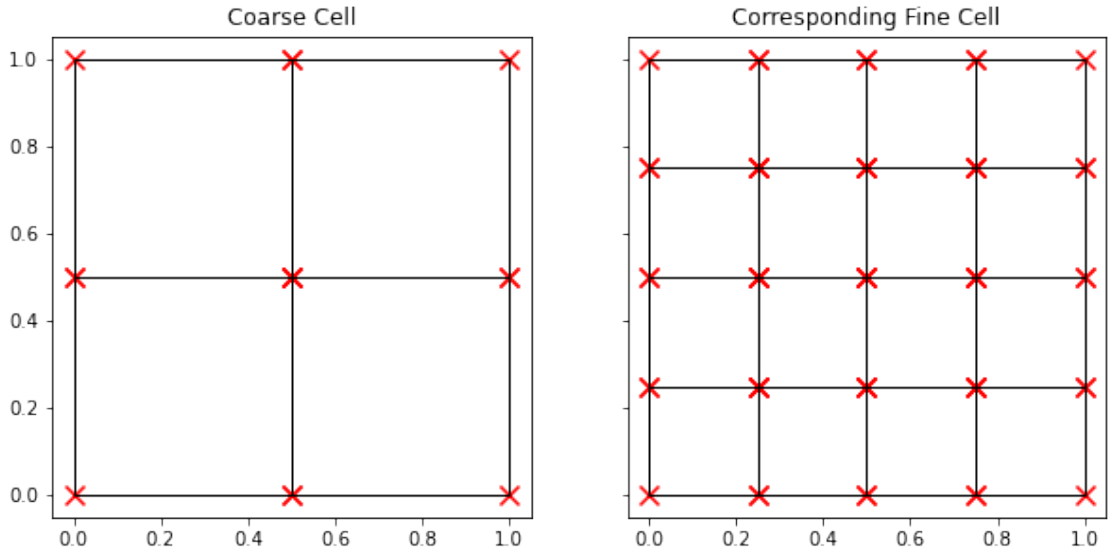
4.1 The interpolation and prolongation operators

As we've discussed, we are using a finite elements discretization for our equation. An advantage of this is that we have a continuous solution over the entire domain. This is quite helpful for definiting the interpolation operator.

Let's look at a single cell and the corresponding fine cells:

[6]:
```
# Take the first and only cell of level 0.
cells = list(geometry.grid.dfs(only_level=0))
c = cells[0]

fig, axs = plt.subplots(1, 2, sharey=True, figsize=(10,10))
plotter.plot_grid(fig, axs[0], grid=c, level=1, vertices=geometry.
 ↪vertices_idx_to_coords, plot_vertices=True)
plotter.plot_grid(fig, axs[1], grid=c, level=2, vertices=geometry.
 ↪vertices_idx_to_coords, plot_vertices=True)
axs[0].axis('square')
axs[0].set_title("Coarse Cell")
axs[1].axis('square')
axs[1].set_title("Corresponding Fine Cell")
```

[6]:
```
Text(0.5, 1.0, 'Corresponding Fine Cell')
```



The left cell in the figure is the coarse cell. The red Xs mark the vertices of each cell.

On the right, you can see the corresponding four cells of the grid with one level more. Note that all vertices in the coarse grid also exist in the fine grid. For these vertices, it is trivial to define the fine grid solution. The other vertices can be found by evaluating the basis functions of the coarse cell at their position. In our case, this corresponds to (bi-)linear interpolation.

Below you can find some examples of iterating over the vertices of a cell. You can also see how to find the global and reference coordinates for each. Note that each vertex has a global index and a data index. Because the unknowns for the Dirichlet vertices are not stored explicitely, both indices are not identical. Before accessing any matrix/vector that has anything to do with the discrete solution, you need to convert the vertex idx to the data idx.

Note that both vertex and data indices are ordered s.t. all indices for a coarse grid also exist in all finer grids.

[7]:
```
# Example of finding vertices:

# Take a random cell
cell = next(geometry.grid.dfs(only_level=1))
print("Found cell with center {center} and size {size}".format(
    center=cell.center,
    size=cell.size))

# Next we find the parent of the cell
parent = cell.parent
print("Its parent has center {center} and size {size}".format(
    center=parent.center,
    size=parent.size))
```

```
# Our cell has a few vertices:
for i, vertex in enumerate(cell.vertices):
    global_coords = geometry.vertices_idx_to_coords[vertex]
    reference_coords = fem.map_to_reference_coordinates(cell, global_coords)
    print(
        "child's {i}th vertex has index {idx}, data index {data_idx}, global␣
 ↪coordinates {global_coords} and reference coordinates {reference_coords}".
 ↪format(
    i=i,
    idx=vertex,
    data_idx=geometry.vertex_idx_to_data_idx[vertex],
    global_coords=global_coords,
    reference_coords=reference_coords))

print()
# The parent also has vertices:
for i, vertex in enumerate(parent.vertices):
    global_coords = geometry.vertices_idx_to_coords[vertex]
    reference_coords = fem.map_to_reference_coordinates(parent, global_coords)
    print(
        "parent's {i}th vertex has index {idx}, data index {data_idx},  global␣
 ↪coordinates {global_coords} and reference coordinates {reference_coords}".
 ↪format(
    i=i,
    idx=vertex,
    data_idx=geometry.vertex_idx_to_data_idx[vertex],
    global_coords=global_coords,
    reference_coords=reference_coords))
# Note that the vertices of both cells have the same reference coordinates.


# Now we want to evaluate the
# We can take the first vertex of the PARENT
first_basis_function_parent = parent.vertices[0]
first_vertex_cell = cell.vertices[0]
coordinates_vertex_cell = np.array(geometry.
 ↪vertices_idx_to_coords[first_vertex_cell])
# Important: We map to the reference coordinates of the PARENT!
reference_coordinates_vertex_cell = fem.map_to_reference_coordinates(parent,␣
 ↪coordinates_vertex_cell)

# and evaluate the basis function of the parent
# Note here that the basis is
print("\nEvaluated first basis function of the parent at position of first␣
 ↪reference cell: {}".format(
    fem.lagrange_2d(discretization.nodes_x, first_basis_function_parent,␣
 ↪reference_coordinates_vertex_cell)))
```

```
# We can also do this with the second basis function and vertex.
# Note that these two have the same coordinates.
# We thus expect the basis function to evaluate to one.
basis_parent = parent.vertices[1]
coords_vertex_cell = fem.map_to_reference_coordinates(parent,
                                                      np.array(geometry.
 ↪vertices_idx_to_coords[cell.vertices[1]]))
print("Evaluated second basis function of the parent at position of second␣
 ↪reference cell: {}".format(
    fem.lagrange_2d(discretization.nodes_x, basis_parent, coords_vertex_cell)))

# Show that vertices of level l are subset of vertices of level l+1
def collect_vertex_ids(geometry, level):
    vertex_ids = set()
    data_ids = set()
    # You can iterate through the cells of a level with a depth-first search.
    for cell in geometry.grid.dfs(only_level=level):
        for vertex in cell.vertices:
            vertex_ids.add(vertex)
            if vertex not in geometry.dirichlet_vertices:
                data_ids.add(geometry.vertex_idx_to_data_idx[vertex])

    return vertex_ids, data_ids

v_0, d_0 = collect_vertex_ids(geometry, 0)
v_1, d_1 = collect_vertex_ids(geometry, 1)
print("Vertices of level 0 are\t\t{},\nvertices of level 1 are\t\t{}".
 ↪format(v_0, v_1))
print("Data indices of level 0 are\t{},\ndata indices of level 1 are\t{}".
 ↪format(d_0, d_1))
```

```
Found cell with center [0.25 0.75] and size 0.5
Its parent has center [0.5 0.5] and size 1.0
child's 0th vertex has index 4, data index 2, global coordinates (0.0, 0.5) and
reference coordinates [-1. -1.]
child's 1th vertex has index 1, data index -1, global coordinates (0.0, 1.0) and
reference coordinates [-1.  1.]
child's 2th vertex has index 5, data index 3, global coordinates (0.5, 0.5) and
reference coordinates [ 1. -1.]
child's 3th vertex has index 6, data index -1, global coordinates (0.5, 1.0) and
reference coordinates [1. 1.]

parent's 0th vertex has index 0, data index 0,  global coordinates (0.0, 0.0)
and reference coordinates [-1. -1.]
parent's 1th vertex has index 1, data index -1,  global coordinates (0.0, 1.0)
and reference coordinates [-1.  1.]
```

```
parent's 2th vertex has index 2, data index 1,  global coordinates (1.0, 0.0)
and reference coordinates [ 1. -1.]
parent's 3th vertex has index 3, data index -1,  global coordinates (1.0, 1.0)
and reference coordinates [1. 1.]

Evaluated first basis function of the parent at position of first reference
cell: 0.5
Evaluated second basis function of the parent at position of second reference
cell: 1.0
Vertices of level 0 are         {0, 1, 2, 3},
vertices of level 1 are         {0, 1, 2, 3, 4, 5, 6, 7, 8}
Data indices of level 0 are     {0, 1},
data indices of level 1 are     {0, 1, 2, 3, 4, 5}
```

4.2 Interpolation

[6 points]

Your task is to create the interpolation matrix. Below are two function templates that you need to adapt.

The first function evaluate_solution should evaluate all basis functions of a cell at a given global cordinate. It should return a dictionary that maps vertices of the cell to the evauated basis functions. Use the functions *fem.map_to_reference_coordinates(cell, coord)* to map the coordinates to the reference coordinates and the function *fem.lagrange_2d(discretization.nodes_x, index, reference_coords)* to evaluate the basis function with index *index*. Note that the vertices are sorted s.t. they are ordered identically to the basis functions when looping through cell.vertices. Furthermore be careful to check whether the vertex is a Dirichlet vertex - in this case, the coefficient has to be zero.

The set of all vertex ids corresponding to the Dirichlet BC is in *geometry.dirichlet_vertices*.

The second function make_interpolation maps from a level_coarse to the next level. The interpolation matrix has the entries (i, j) where i are the data indices correspond tovertices of the coarse grid and j are the coefficients of the hat basis. The algorithm for building up the interpolation matrix is quite simple:

- Loop through all cells
- Find their parent with cell.parent
- Loop over all vertices (in cell.vertices)
- Find their indices with geometry.vertices_idx_to_coords[vertex_id]
- Use evaluate_solution to find the coefficients of the parent's basis functions and set the entry in the matrix.

You basically have to loop through all cells of a given level, and then use the function evaluate_solution to find the coefficients. You can also refer to worksheet 9, which defines the interpolation operator for the 1D-case - the 2D-algorithm is identical. Note that you should be careful to use *geometry.vertex_idx_to_data_idx* to convert from the vertex to the data index, where appropiate. You can find the number of data points per level in the array *geometry.data_size_per_level[level]*.

Only return sparse matrices. Otherwise you will run out of memory for larger levels.

Remark: The algorithm basically performs bi-linear interpolation and can also be implemented as a matrix-vector product. For this worksheet, use the algorithm defined above.

Task: Compute the prolongation/interpolation matrix.

[8]:
```
def evaluate_solution(geometry, discretization, cell, coord):
    # Get the parent of the cell
    parent = cell.parent
    # Get the reference coordinates of the coord in the parent cell
    reference_coordinates = fem.map_to_reference_coordinates(parent, coord)
    # Declare dictionary to store evaluated basis functions at each coordinate
    evaluated_cell_basis = dict()
    # Loop through the parent's vertices
    for i, vertex in enumerate(parent.vertices):
        # Ignore Dirichlet vertices
        if vertex not in geometry.dirichlet_vertices:
            # Calculate the data index
            data_idx = geometry.vertex_idx_to_data_idx[vertex]
            # Populate the dictionary with the evaluation of the basis function␣
↪with the respective
            # data index
            evaluated_cell_basis[data_idx] = fem.lagrange_2d(discretization.
↪nodes_x, i, reference_coordinates)
    # Return dictionary
    return evaluated_cell_basis

def make_interpolation(geometry, level_coarse):
    # Deduce fine level and set up the discritization for the fine level
    level_fine = level_coarse + 1
    discretization = fem.Discretization(geometry, level_fine)

    # Set up the dimensions and declare the interporation matrix as a sparse␣
↪matrix
    number_of_vertices_coarse = geometry.data_size_per_level[level_coarse]
    number_of_vertices_fine = geometry.data_size_per_level[level_fine]
    interpolation = sp.lil_matrix((number_of_vertices_fine,␣
↪number_of_vertices_coarse), dtype=np.float64)

    # Abbreviation: ATMT = *AVOIDING *TRAVERSING THE VERTICES *MULTIPLE *TIMES␣
↪WITHIN THE SAME PARENT
    # These checks avoid calculting basis functions for the same vertex within␣
↪same parent
    # Improves problem set up time significantly for large grids

    # ATMT: Declare current parent
    current_parent = None
    # Loop through all the fine level cells
```

```
    for cell_fine in geometry.grid.dfs(only_level=level_fine):
        # ATMT: Get the parent of current cell
        cell_parent = cell_fine.parent
        # ATMT: If it is a new parent empty the list␣
↪traversed_vertices_in_same_parent and set current parent
        # ATMT: The size of the list would be maximum 9
        if cell_parent is not current_parent:
            traversed_vertices_in_same_parent = []
            current_parent = cell_parent
        # Loop through all the vertices of the fine level cell
        for vertex in cell_fine.vertices:
            # Ignore Direchlet boundary vertices and,
            # ATMT: Ignore if the vetex is already traversed within same parent
            if vertex not in geometry.dirichlet_vertices and vertex not in␣
↪traversed_vertices_in_same_parent:
                # ATMT: Append the list traversed_vertices_in_same_parent with␣
↪the vertex
                traversed_vertices_in_same_parent.append(vertex)
                # Get the global coordinates of the vertex
                global_coordinates = np.array(geometry.
↪vertices_idx_to_coords[vertex])
                # Get the dictionary of evaluated basis functions for the␣
↪global coordinate
                evaluated_cell_basis = evaluate_solution(geometry,␣
↪discretization, cell_fine, global_coordinates)
                # Get the data index of the fine level vertex
                fine_data_idx = geometry.vertex_idx_to_data_idx[vertex]
                # Populate the matrix according to the fine level data index␣
↪and the
                # coarse level data index from the dictionary
                interpolation[fine_data_idx,list(evaluated_cell_basis.keys())]␣
↪=  list(evaluated_cell_basis.values())
    # Return the interpolation matrix in the Compressed Sparse Column format
    return interpolation.tocsc()

level_coarse = 1

interpolation = make_interpolation(geometry, level_coarse=level_coarse)
print("Interpolation Matrix\n " ,interpolation.todense())
print("Number of non-zero values in Interpolation Matrix",np.
↪count_nonzero(interpolation.todense()))
```

```
Interpolation Matrix
  [[1.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.  ]
 [0.   0.   1.   0.   0.   0.  ]
 [0.   0.   0.   1.   0.   0.  ]
```

```
 [0.   0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.   1.  ]
 [0.   0.   0.5  0.   0.   0.  ]
 [0.   0.   0.25 0.25 0.   0.  ]
 [0.   0.   0.   0.5  0.   0.  ]
 [0.   0.   0.5  0.5  0.   0.  ]
 [0.   0.   0.   0.25 0.25 0.  ]
 [0.   0.   0.   0.   0.5  0.  ]
 [0.   0.   0.   0.5  0.5  0.  ]
 [0.5  0.   0.5  0.   0.   0.  ]
 [0.25 0.   0.25 0.25 0.   0.25]
 [0.   0.   0.   0.5  0.   0.5 ]
 [0.5  0.   0.   0.   0.   0.5 ]
 [0.   0.25 0.   0.25 0.25 0.25]
 [0.   0.5  0.   0.   0.5  0.  ]
 [0.   0.5  0.   0.   0.   0.5 ]]
Number of non-zero values in Interpolation Matrix 35
```

4.3 Restriction

[2 points]

The restriction matrix is the transpose of the interpolation matrix. Implement this as the function make_restricion, again making sure that the matrix is sparse.

[9]:
```
def make_restriction(geometry, level_coarse):
    # Generate the interpolation matrix from the make_interpolation function
    interpolation = make_interpolation(geometry, level_coarse=level_coarse)
    # Deduce the restriction matrix by taking the transpose of the␣
 ↪interpolation matrix
    restriction = interpolation.T
    # Return in Compressed Sparse Column format
    return restriction.tocsc()

make_restriction(geometry, level_coarse=level_coarse).todense()
```

[9]:
```
matrix([[1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
         0.  , 0.  , 0.  , 0.5 , 0.25, 0.  , 0.5 , 0.  , 0.  , 0.  ],
        [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
         0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.25, 0.5 , 0.5 ],
        [0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.5 , 0.25, 0.  , 0.5 ,
         0.  , 0.  , 0.  , 0.5 , 0.25, 0.  , 0.  , 0.  , 0.  , 0.  ],
        [0.  , 0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.25, 0.5 , 0.5 ,
         0.25, 0.  , 0.5 , 0.  , 0.25, 0.5 , 0.  , 0.25, 0.  , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
         0.25, 0.5 , 0.5 , 0.  , 0.  , 0.  , 0.  , 0.25, 0.5 , 0.  ],
        [0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  , 0.  , 0.  ,
         0.  , 0.  , 0.  , 0.  , 0.25, 0.5 , 0.5 , 0.25, 0.  , 0.5 ]])
```

5 Coarse Grid Operator

[6 points]

Next, we use the Galerkin construction of the coarse grid operators. We define the coarse grid operators as

$$A^{2h} = R^{h \to 2h} A^h P^{2h \to ih}$$

where A^{2h} is the operator on a grid that's one level smaller than the operator A^h.

$P^{h \to 2h}$ is the prologation/interpolaton operator that maps a vector from a grid to a grid of the next level and $R^{h \to 2h}$ is the restriction operator that maps to the next coarser grid.

The data structure is already defined below. You need to fill in the blanks and initialize the variables

```
coarse_to_fine
fine_to_coarse
stiffness
```

You can initialize the stiffness matrices by going from the finest grid backwards and computing the coarser operator from the previously computed matrix.

The output for level_min=1, level_max=3 should look like this (you can ignore the specific sparse matrix format, as long as the matrices are sparse):

```
storage.fine_to_coarse
([<2x6 sparse matrix of type '<class 'numpy.float64'>'
    with 8 stored elements in Compressed Sparse Row format>,
  <6x20 sparse matrix of type '<class 'numpy.float64'>'
    with 35 stored elements in Compressed Sparse Row format>,
  <20x72 sparse matrix of type '<class 'numpy.float64'>'
    with 143 stored elements in Compressed Sparse Row format>]

storage.coarse_to_fine
[<6x2 sparse matrix of type '<class 'numpy.float64'>'
    with 8 stored elements in Compressed Sparse Column format>,
  <20x6 sparse matrix of type '<class 'numpy.float64'>'
    with 35 stored elements in Compressed Sparse Column format>,
  <72x20 sparse matrix of type '<class 'numpy.float64'>'
    with 143 stored elements in Compressed Sparse Column format>]

storage.stiffness
[<6x6 sparse matrix of type '<class 'numpy.float64'>'
    with 28 stored elements in Compressed Sparse Row format>,
  <20x20 sparse matrix of type '<class 'numpy.float64'>'
    with 130 stored elements in Compressed Sparse Row format>,
  <72x72 sparse matrix of type '<class 'numpy.float64'>'
    with 550 stored elements in Compressed Sparse Column format>]
```

The coarse grid operators that you get from:

- direct discretization on the coarse grid

- Galerkin construction

should be identical up to a constant.

[10]:
```
class MultigridStorage:
    def __init__(self, discretization, level_min, level_max, stiffness):
        self.level_min = level_min
        self.level_max = level_max

        self.geometry = discretization.geometry
        self.number_of_levels = (level_max - level_min + 1)

        ## Storage for Interpolation Matrices
        self.coarse_to_fine = []
        ## Storage for Restriction Matrices
        self.fine_to_coarse = []
        ## Storage for Stiffness Matrices
        self.stiffness = [stiffness]

        ## Populate the storage with interpolation and restriction matrices at␣
↪each level
        for level in range(level_min,level_max+1):
            ## Compute Interpolation matrix
            interpolation_matrix = make_interpolation(self.geometry, level-1)
            ## Store Interpolation matrix directly
            self.coarse_to_fine.append(interpolation_matrix)

            ## Transpose already computed Interpolation matrix to get␣
↪Restriction matrix and store it
            ## This is to avoid calling 'make_interpolation' function from␣
↪within 'make_restriction'
            ## (which is the most expensive function to evaluate, in our case␣
↪atleast)
            self.fine_to_coarse.append(interpolation_matrix.T)
            ## Compute Restriction Matrix and store (Not used)
            # self.fine_to_coarse.append(make_restriction(self.
↪geometry,level-1))

        ## Populate the stiffness matrices in the reverse order from the finest␣
↪to the coarsest
        for level in reversed(range(level_min, level_max)):
            ## Calculate index corresponding to level in storage
            index_level = level - self.level_min + 1
            ## Calculate Coarse (level - 1) Matrix from fine (level) stiffness␣
↪matrix and corresponding
            ## interpolation and restriction matrix
```

```
            next_stiffness = self.fine_to_coarse[index_level] @ self.
 ↪stiffness[0] \
                             @ self.coarse_to_fine[index_level]
            ## Insert the coarse level stiffness matrix to the front of list␣
 ↪(Tried 'deque appendleft()' too but
            ## not too much perfomance upgrade in place of 'insert' operation)
            self.stiffness.insert(0, next_stiffness)

level = 3
geometry = Geometry(level=level)
eval_k = lambda x, y: x + y + 0.001
discretization = fem.Discretization(geometry, level=level, eval_k=eval_k)
stiffness = discretization.setup_stiffness()
rhs = discretization.setup_rhs()
sol = splinalg.linsolve.spsolve(stiffness, rhs)

storage = MultigridStorage(discretization, level_min=1, level_max=level,␣
 ↪stiffness=stiffness)

print(storage.fine_to_coarse,"\n")
print(storage.coarse_to_fine, "\n")
print(storage.stiffness)
```

```
Level 0 has     4 vertices
Level 1 has     5 vertices
Level 2 has     16 vertices
Level 3 has     56 vertices
[<2x6 sparse matrix of type '<class 'numpy.float64'>'
        with 8 stored elements in Compressed Sparse Row format>, <6x20 sparse
matrix of type '<class 'numpy.float64'>'
        with 35 stored elements in Compressed Sparse Row format>, <20x72 sparse
matrix of type '<class 'numpy.float64'>'
        with 143 stored elements in Compressed Sparse Row format>]

[<6x2 sparse matrix of type '<class 'numpy.float64'>'
        with 8 stored elements in Compressed Sparse Column format>, <20x6 sparse
matrix of type '<class 'numpy.float64'>'
        with 35 stored elements in Compressed Sparse Column format>, <72x20
sparse matrix of type '<class 'numpy.float64'>'
        with 143 stored elements in Compressed Sparse Column format>]

[<6x6 sparse matrix of type '<class 'numpy.float64'>'
        with 28 stored elements in Compressed Sparse Row format>, <20x20 sparse
matrix of type '<class 'numpy.float64'>'
        with 130 stored elements in Compressed Sparse Row format>, <72x72 sparse
matrix of type '<class 'numpy.float64'>'
        with 550 stored elements in Compressed Sparse Column format>]
```

6 Smoother

[2 points]

We've seen how we can create a linear system $Ax = b$ with our grid and dicretization. How can we solve it?

The next step towards multigrid is defining the smoother. Here, we want to use a simple Jacobi relaxation. We split the matrix A into a diagonal and an off-diagonal part as

$$A = D + O,$$

we can then write one Jacobi update as

$$x^{n+1} = D^{-1}(b - Ox^n).$$

Task: Implement this. Either use the row-wise method or use sparse linear algebra methods from scipy. Never build up a dense matrix! You are free to use the method scipy.sparse.diags (or sp.diags here) to extract the diagonal part of A.

[11]:
```python
def jacobi_relaxation(A, x, b, num_it):
    ## Initial guess
    x_new = x
    ## Extract diagonal elements in a list
    D = A.diagonal()
    ## Sparse Inverse Diagonal Matrix
    Dinv = sp.diags(1./D)
    ## Sparse Diagonal Matrix
    D = sp.diags(D)

    ## Jacobi Relaxation
    for i in range(num_it):
        x_new = Dinv @ (b-(A-D) @ x)
        # Update x with x_new
        x = x_new
    return x

def compute_residual(A, x, b, norm = True):
    """Computes Residual.

    Parameters
    ----------
    A : `numpy.ndarray`, (N, N)
    x : `numpy.ndarray`, (N,)
    y : `numpy.ndarray`, (N,)

    Returns
    -------
    Residual : `float`
        if norm = True
```

```
    Residual : `numpy.ndarray`, (N,)
        if norm = False
    """
    # Return norm of residual if norm = True
    if (norm):
        return np.linalg.norm(b - A @ x)
    # Return the residual vector otherwise
    return  b - A @ x
```

7 V-Cycle

[12 points]

We use the following recursive algorithm do define one v-cycle (see A multigrid tutorial Chapter 3 or lecture for more details). The vcycle operator V^h for a grid with spacing h, initial guess v^h and right-hand-side f^h is

$$v^h = V^h(v^h, f^h)$$

1. Relax s_1 times on $A^h u^h = f^h$
2. If already coarsest grid, go to step 5 (no further iteration possible).
3. Else:

   (Setup rhs for residual equation)

   $$f^{2h} = R^{2h \to h}(f^h - A^h v^h)$$

   (Set initial guess for update to zero)

   $$v^{2h} = 0$$

   (Call vcycle recursively.)

   $$v^{2h} = V^{2h}(v^{2h}, f^{2h})$$

4. Correct

   $$v^h = v^h + P^{2h \to h} v^{2h}$$

5. Relax s_2 times on $A^h u^h = f^h$ with initial guess v^h.

We have defined all necessary matrices and the smoother in the steps before. The only part left is combining all loose parts into one coherent algorithm.

We choose $s_1 = 5$, $s_2 = 5$.

Task: Implement this. You can use the MultigridStorage that you've defined earlier to access the grid transfer matrices and the coarse grid operators. Show the solution that you get. Use the code below to compare the v-cycle and Jacobi relaxation.

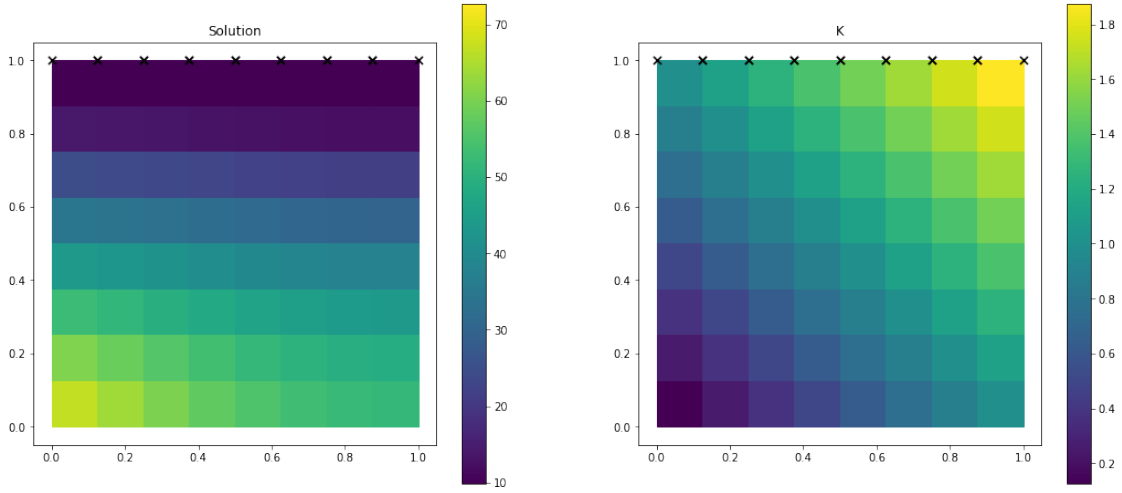
[12]:
```
def v_cycle(storage, level, rhs, unknowns):

    ## Calculate the index in storage for current level
    index_level = level - storage.level_min

    ## Presmoothing by calling Jacobi relaxation 5 times for A_h, u_h , f_h
    if(level == storage.level_max):
        ## Hardcoding S1 = 5 since it's given
        unknowns = jacobi_relaxation(A=storage.stiffness[index_level],␣
↪x=unknowns, b=rhs, num_it=5)
        # Store initial_solution for correction at the end of the vcycle
        initial_solution = unknowns

    ## Unknowns in the next coarser grid
    unknowns_coarse = np.zeros(storage.fine_to_coarse[index_level].shape[0])

    ## Check if the grid is not at the coarsest level
    if(level > storage.level_min):
        ## Calculate Residual
        residual = compute_residual(A=storage.stiffness[index_level],␣
↪x=unknowns, b=rhs, norm=False )
        ## Restrict to the current level
        rhs_coarse = storage.fine_to_coarse[index_level] @ residual
        ## Recursively Call vcycle
        unknowns = v_cycle(storage, level - 1, rhs_coarse, unknowns_coarse)

    #Solve if at the coarsest level with Jacobi smoothing again
    if(level == storage.level_min):
        ## Hardcoding S2 = 5 since it's given
        unknowns = jacobi_relaxation(A=storage.stiffness[index_level],␣
↪x=unknowns, b=rhs, num_it=5)

    ## Interpolation if level is less than finest level
    if (level < storage.level_max):
        ## Unknowns in the next finer grid
        unknowns_fine = np.zeros(storage.coarse_to_fine[index_level+1].shape[0])
        ## Interpolation
        unknowns_fine = storage.coarse_to_fine[index_level+1] @ unknowns

    # Final steps at the finest level
    if(level == storage.level_max):
        ## Unknowns in the finest grid
        unknowns_fine = np.zeros(storage.coarse_to_fine[index_level].shape[0])
        ## Correction of error at the finest level
        unknowns_fine = unknowns + initial_solution
        ## Post-smoothing by calling Jacobi relaxation 5 times for A_h, u_h ,␣
↪f_h on the finest level
```

```
        ## Hardcoding S2 = 5 since it's given
        unknowns_fine = jacobi_relaxation(A=storage.stiffness[index_level],␣
↪x=unknowns_fine, b=rhs, num_it=5)

    return unknowns_fine
```

[13]:
```
# Zero initial guess
unknowns = np.zeros(geometry.data_size_per_level[storage.level_max])
total_time = 0.0
for cycle in range(10):
    start = time.perf_counter()
    unknowns = v_cycle(storage, level=storage.level_max, rhs=rhs,␣
↪unknowns=unknowns)
    end = time.perf_counter()
    total_time += end - start
    print("Residual after {} iterations = {},\tError = {}".
          format(
          (cycle+1),
          compute_residual(stiffness, unknowns, rhs),
          np.linalg.norm(sol - unknowns)
          ))

print("Total Time: ", total_time)
```

```
Residual after 1 iterations = 0.06796741277153114,      Error =
53.78268140644855
Residual after 2 iterations = 0.010597805141273512,     Error =
8.225130288924467
Residual after 3 iterations = 0.0016556594225339246,    Error =
1.2678353594638476
Residual after 4 iterations = 0.0002576884259633 1475,  Error =
0.1959205774308827
Residual after 5 iterations = 3.999728808015285e-05,    Error =
0.030302391041618984
Residual after 6 iterations = 6.19884521701232e-06,     Error =
0.004688214104780366
Residual after 7 iterations = 9.599630583425155e-07,    Error =
0.0007254127509950461
Residual after 8 iterations = 1.4860349388028395e-07,   Error =
0.00011224824045475793
Residual after 9 iterations = 2.29995165770849e-08,     Error =
1.736919507467417e-05
Residual after 10 iterations = 3.559309600411759e-09,   Error =
2.6877065507302736e-06
Total Time:  0.12977474000126676
```

[14]:
```
plot_solution(geometry, discretization, unknowns)
```

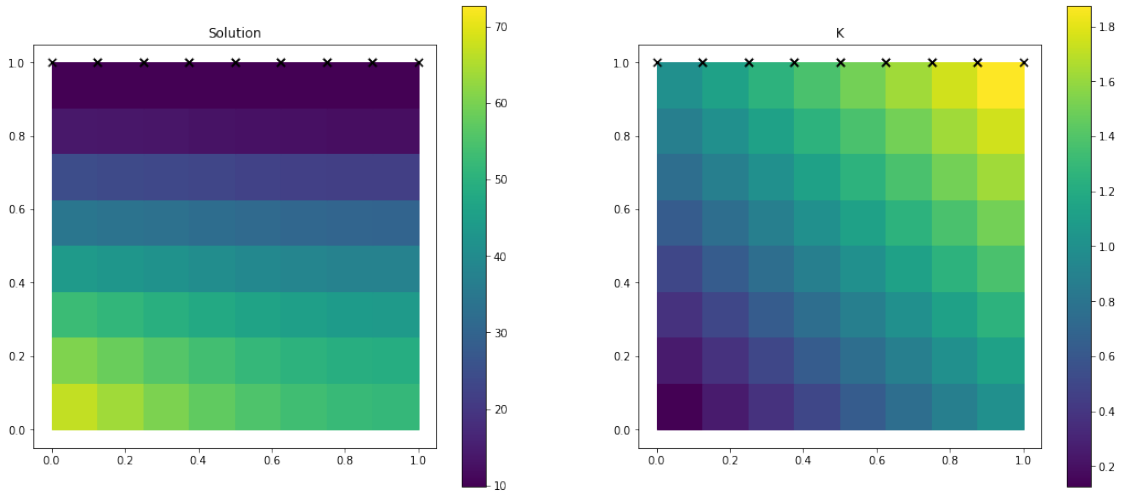


[15]:
```
unknowns = np.zeros(geometry.data_size_per_level[storage.level_max])
total_time = 0.0
for it in range(10):
    start = time.perf_counter()
    unknowns = jacobi_relaxation(A=stiffness, x=unknowns, b=rhs, num_it=100)
    end = time.perf_counter()
    total_time += end - start
    print("Residual after {} \titerations = {},\tError = {}".
          format(
          (it+1) * 100,
          compute_residual(stiffness, unknowns, rhs),
          np.linalg.norm(sol - unknowns)
          ))
print("Total Time: ", total_time)
```

```
Residual after 100      iterations = 0.052904352579806836,      Error =
40.69796300381208
Residual after 200      iterations = 0.0061907758704244555,     Error =
4.7618799621785834
Residual after 300      iterations = 0.0007243800651490803,     Error =
0.557185498285041
Residual after 400      iterations = 8.47593986450789e-05,      Error =
0.06519603456160535
Residual after 500      iterations = 9.917660636429444e-06,     Error =
0.007628559852700098
Residual after 600      iterations = 1.160461188527307e-06,     Error =
0.0008926144943846623
Residual after 700      iterations = 1.3578506229382944e-07,    Error =
0.00010444443660629232
```

```
Residual after 800      iterations = 1.588815137876132e-08,     Error =
1.222099849910344e-05
Residual after 900      iterations = 1.859065164891264e-09,     Error =
1.4299738481308959e-06
Residual after 1000     iterations = 2.1752786358978566e-10,    Error =
1.6732067075295299e-07
Total Time:  0.36303917599889246
```

[16]:
```
plot_solution(geometry, discretization, unknowns)
```



7.1 Analyzing the convergence

[6 points]

Task We want to compare the convergnce properties of our smoother with the v-cycle algorithm. Answer the following questions:

- How many iterations are needed to get a residual lower than 1e-6?
- How many prolongations, restrictions, and smoother steps are needed for one/all multigrid cycles?

For the first one, write the code below, for the second one, write your answer in the following text field.

Answer:

For one multigrid cycle:

- Number of prolongations = level_max - level_min
- Number of restrictions = level_max - level_min
- Number of smoother steps = $s_1 + 2s_2 = 5 + 2 \times 5 = 15$

For all multigrid cycles n ($n = 7$ in the below example):

- Number of prolongations = (level_max - level_min)$n = (3-1) \times 7 = 14$
- Number of restrictions = (level_max - level_min)$n = (3-1) \times 7 = 14$
- Number of smoother steps = $(s_1 + 2s_2)n = (5 + 2 \times 5) \times 7 = 105$

[17]:
```
## Compute iterations for Jacobi
unknowns = np.zeros(geometry.data_size_per_level[storage.level_max])

# Initialize residual
residual = float('inf')
iterations = 0

while residual > 1e-6:
    unknowns = jacobi_relaxation(A=stiffness, x=unknowns, b=rhs, num_it=1)
    residual = compute_residual(stiffness, unknowns, rhs)
    iterations += 1
print("Jabcobi needs {} iterations to get a residual lower than 1e-6".
 ↪format(iterations))
```

```
Jabcobi needs 607 iterations to get a residual lower than 1e-6
```

[18]:
```
## Compute iterations for v-cycle
unknowns = np.zeros(geometry.data_size_per_level[storage.level_max])

# Initialize residual
residual = float('inf')
iterations = 0

while residual > 1e-6:
    unknowns = v_cycle(storage, level=storage.level_max, rhs=rhs, unknowns =␣
 ↪unknowns)
    residual = compute_residual(stiffness, unknowns, rhs)
    iterations += 1
print("V-cycle needs {} cycles to get a residual lower than 1e-6".
 ↪format(iterations))
```

```
V-cycle needs 7 cycles to get a residual lower than 1e-6
```

7.2 Analyizing the time complexity

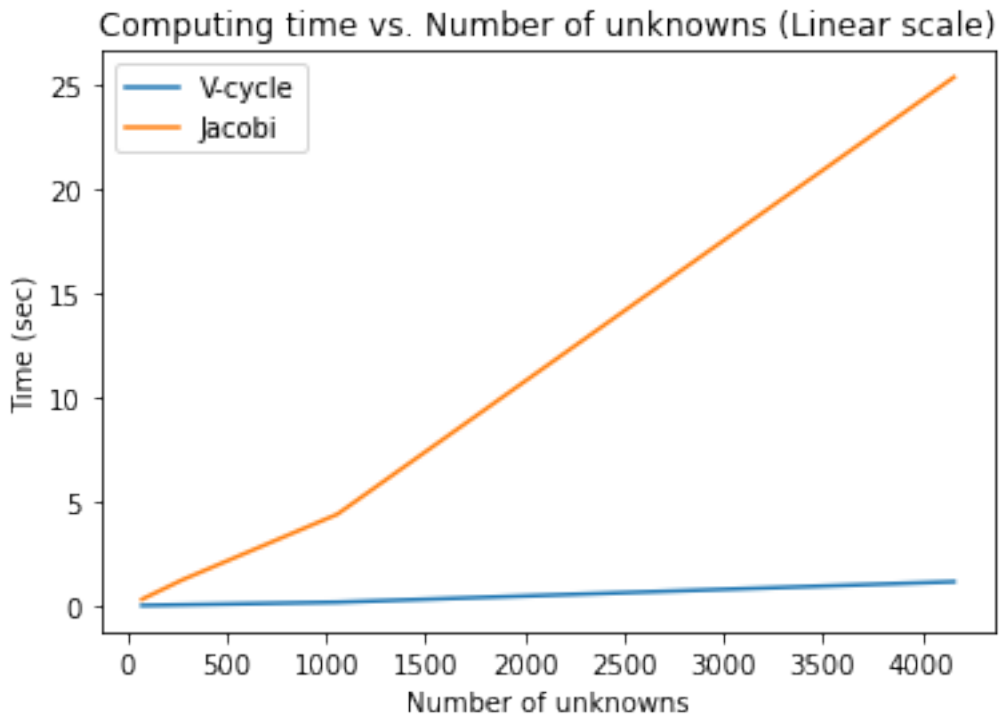
[19]:
```
max_plot_levels = 6
plot_start_level = 3
geometry = Geometry(level=max_plot_levels)

jtime = np.zeros([max_plot_levels - plot_start_level + 1])
vtime = np.zeros([max_plot_levels - plot_start_level + 1])
levellist = np.zeros_like(vtime)
jit = np.zeros_like(jtime)
```

```
vit = np.zeros_like(vtime)
Nit = np.zeros_like(vtime)

for it,i in enumerate(np.arange(plot_start_level, max_plot_levels + 1)):
    eval_k = lambda x, y: x + y + 0.001
    discretization = fem.Discretization(geometry, level=i, eval_k=eval_k)
    stiffness = discretization.setup_stiffness()
    rhs = discretization.setup_rhs()
    Nit[it] = stiffness.shape[0]

    storage = MultigridStorage(discretization, level_min=1 , level_max=i,␣
↪stiffness=stiffness)

    print("\nDiscretization level: ", i)
    levellist[it] = i

    ### Jacobi
    unknowns = np.zeros(geometry.data_size_per_level[storage.level_max])
    residual = float('inf')
    iterations = 0
    start = time.perf_counter()
    while residual > 1e-6:
        unknowns = jacobi_relaxation(A=stiffness, x=unknowns, b=rhs, num_it=1)
        residual = compute_residual(stiffness, unknowns, rhs)
        iterations += 1
    end = time.perf_counter()
    jtime[it] = end-start
    jit[it] = iterations - 1
    print("Jacobi needs {} iterations to get a residual lower than 1e-6".
↪format(iterations))

    ### V-cycle
    unknowns = np.zeros(geometry.data_size_per_level[storage.level_max])
    residual = float('inf')
    iterations = 0
    start = time.perf_counter()
    while residual > 1e-6:
        unknowns = v_cycle(storage, level=storage.level_max, rhs=rhs, unknowns␣
↪= unknowns)
        residual = compute_residual(stiffness, unknowns, rhs)
        iterations += 1
    end = time.perf_counter()
    vtime[it] = end-start
    vit[it] = iterations - 1
    print("V-cycle needs {} cycles (Total Jacobi smoothing iterations = {}) to␣
↪get a residual lower than 1e-6".
          format(iterations, 15 * iterations))
```

```
Level 0 has     4 vertices
Level 1 has     5 vertices
Level 2 has     16 vertices
Level 3 has     56 vertices
Level 4 has     208 vertices
Level 5 has     800 vertices
Level 6 has     3136 vertices

Discretization level:  3
Jacobi needs 607 iterations to get a residual lower than 1e-6
V-cycle needs 7 cycles (Total Jacobi smoothing iterations = 105) to get a
residual lower than 1e-6

Discretization level:  4
Jacobi needs 2306 iterations to get a residual lower than 1e-6
V-cycle needs 14 cycles (Total Jacobi smoothing iterations = 210) to get a
residual lower than 1e-6

Discretization level:  5
Jacobi needs 8717 iterations to get a residual lower than 1e-6
V-cycle needs 36 cycles (Total Jacobi smoothing iterations = 540) to get a
residual lower than 1e-6

Discretization level:  6
Jacobi needs 32820 iterations to get a residual lower than 1e-6
V-cycle needs 152 cycles (Total Jacobi smoothing iterations = 2280) to get a
residual lower than 1e-6
```

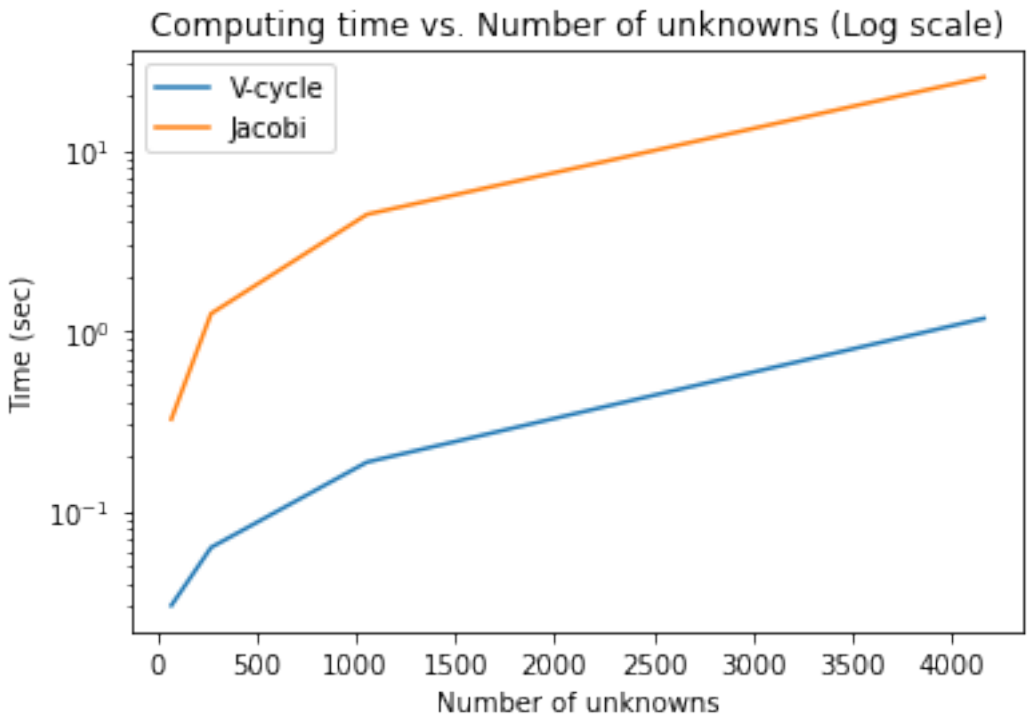
[20]:
```
plt.plot(Nit, vtime, label= "V-cycle")
plt.plot(Nit, jtime, label= "Jacobi")
plt.legend()
plt.xlabel("Number of unknowns")
plt.ylabel("Time (sec)")
plt.title("Computing time vs. Number of unknowns (Linear scale)")
# plt.savefig('linTime.png', dpi=300)
```

[20]:
```
Text(0.5, 1.0, 'Computing time vs. Number of unknowns (Linear scale)')
```



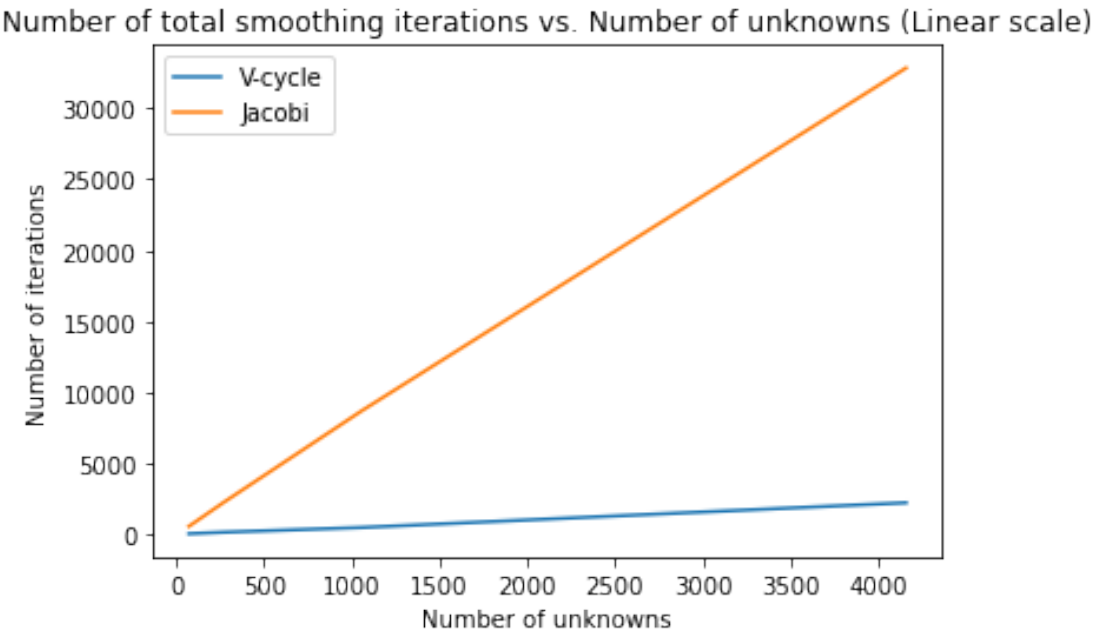
[21]:
```
plt.plot(Nit, vtime, label= "V-cycle")
plt.plot(Nit, jtime, label= "Jacobi")
plt.yscale('log')
plt.legend()
plt.xlabel("Number of unknowns")
plt.ylabel("Time (sec)")
plt.title("Computing time vs. Number of unknowns (Log scale)")
# plt.savefig('logTime.png', dpi=300)
```

[21]:
```
Text(0.5, 1.0, 'Computing time vs. Number of unknowns (Log scale)')
```



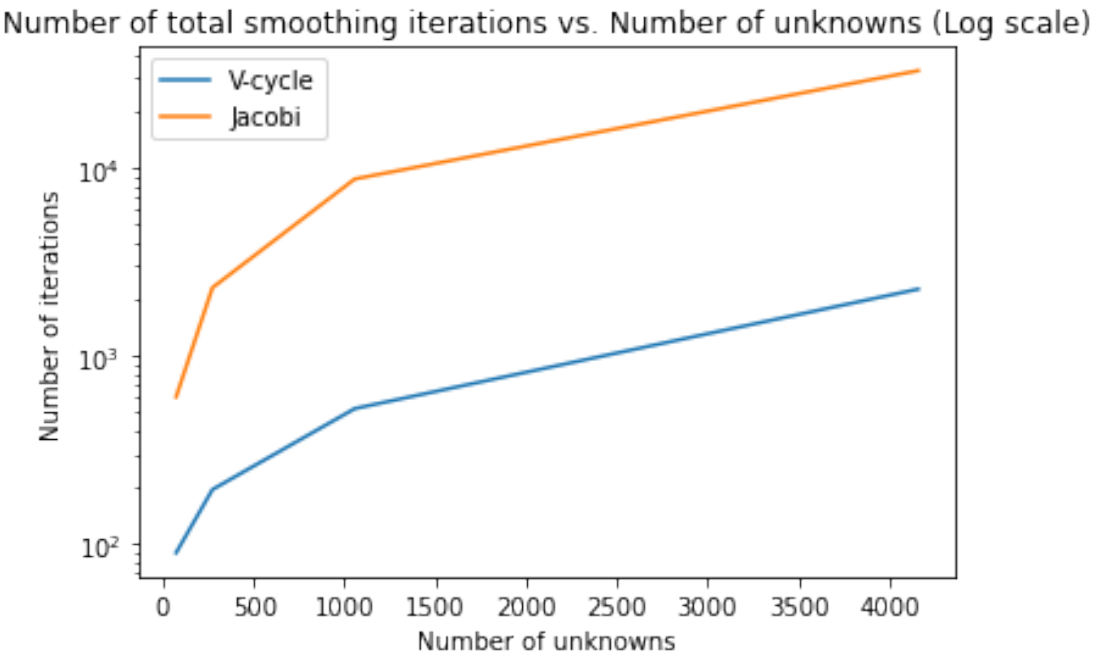
[22]:
```
# Plotting the actual number of Jacobi smoothing iterations done within all␣
 ↪vcycles
plt.plot(Nit, 15 * vit, label= "V-cycle")
plt.plot(Nit, jit, label= "Jacobi")
plt.legend()
plt.xlabel("Number of unknowns")
plt.ylabel("Number of iterations")
plt.title("Number of total smoothing iterations vs. Number of unknowns (Linear␣
 ↪scale)")
# plt.savefig('linIter.png', dpi=300)
```

[22]:
```
Text(0.5, 1.0, 'Number of total smoothing iterations vs. Number of unknowns
(Linear scale)')
```



[23]:
```
# Plotting the actual number of Jacobi smoothing iterations done within all␣
 ↪vcycles
plt.plot(Nit, 15 * vit, label= "V-cycle")
plt.plot(Nit, jit, label= "Jacobi")
plt.yscale('log')
plt.legend()
plt.xlabel("Number of unknowns")
plt.ylabel("Number of iterations")
plt.title("Number of total smoothing iterations vs. Number of unknowns (Log␣
 ↪scale)")
# plt.savefig('logIter.png', dpi=300)
```

[23]:
```
Text(0.5, 1.0, 'Number of total smoothing iterations vs. Number of unknowns (Log
scale)')
```



7.3 Conclusion

From the above plots it can be seen that there is a significant improvement in the Multigrid method with Vcycles compared to using Jacobi solver only. Also, the number of Vcycles in Multigrid method scales linearly with the number of unknowns