

# The 2DG Code

version 1.0

Python Version

## 1 Data Structures

### 1.1 The mesh data structure

We will describe the `mesh` data structure with the help of the following (coarse) triangular mesh for the unit circle:

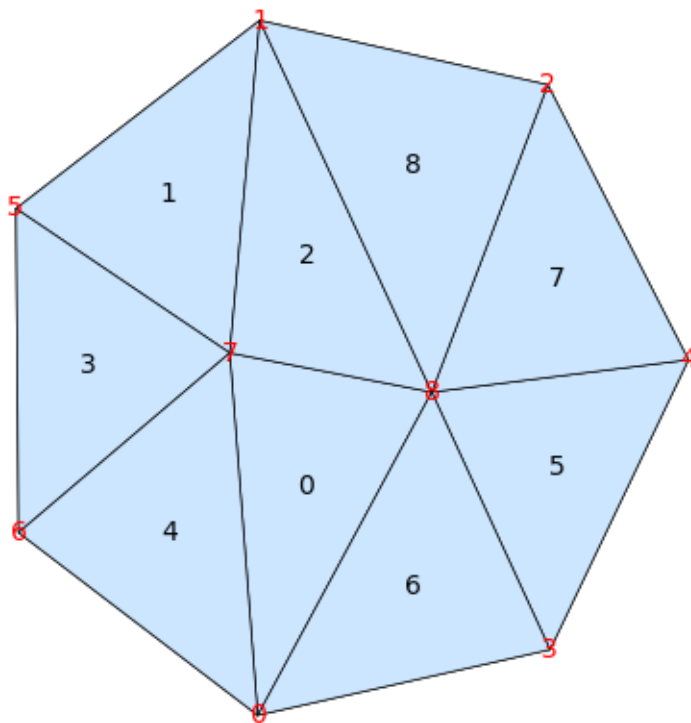


Figure 1: Example of a triangular mesh

The `mesh` data structure contains `numpy` arrays with the following information:

- **The triangular linear mesh.** This mesh consists of `np` vertices and `nt` triangles. For the example in figure 1 we have `np = 9` and `nt = 9`.

`mesh.p (np,2):` x and y coordinates of vertices in triangulation for simplicial mesh

```
>> mesh.p
[[-0.21291331 -0.97707109]
 [-0.20746732  0.97824195]
 [ 0.60383562  0.79710886]
 [ 0.61012115 -0.79230814]
 [ 0.999729    0.02327903]
 [-0.89358373  0.44889655]
 [-0.886458    -0.46280905]
 [-0.29161449  0.04252226]
 [ 0.27702361 -0.06758347]]
```

`mesh.t (nt,3):` Element vertices for simplicial mesh (numbered counterclockwise)

```
>> mesh.t
[[8 7 0]
 [7 1 5]
 [1 7 8]
 [6 7 5]
 [7 6 0]
 [8 3 4]
 [3 8 0]
 [2 8 4]
 [2 1 8]]
```

- **Face and element connectivity information.** Here, `nf` is the number of faces (or edges in 2D) and can be calculated as  $nf = (3*nt + nb)/2$ , where `nb` is the number of boundary edges. For our example `nb = 7`, so `nf = 17`. We introduce two arrays: `mesh.f (nf,4)` and `mesh.t2f (nt,3)`.

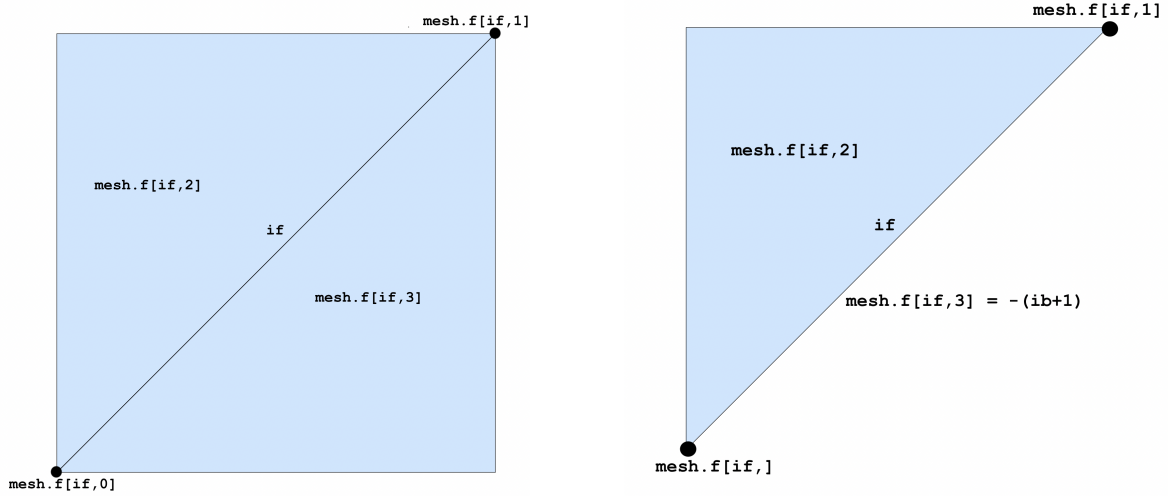


Figure 2: Definition of an interior face `if` (left), and a boundary face `if` on boundary `ib`.

A face is defined as the segment joining two vertices in the triangular mesh. `mesh.f[:, [0,1]]` are the indices of the two vertices in the mesh. `mesh.f[:, 2]` is the index of the triangle to

the left of the edge (when walking from `mesh.f[:,0]` to `mesh.f[:,1]`). For interior edges, `mesh.f[:,3]` is the index of the triangle to the right of the edge. In addition, for boundary edges, `mesh.f[:,3]` is the *negative* of the (boundary indicator plus one). For the case of the circle we only have one boundary type and hence for these edges `mesh.f[:,3] = -1` (since here there is only one boundary type equal to 0 - using python's zero based convention). Note that this convention implies that all the boundary edges are oriented counterclockwise. Finally, for computational efficiency, the edges are ordered so that all interior edges are placed first and the boundary edges are last in the edge list.

`mesh.f (nf,4):`      `mesh.f[:,[0,1]]` are the indices of the two vertices in the mesh.  
                          `mesh.f[:,2]` is the index of the triangle to the left of the edge  
                          (when walking from `mesh.f[:,0]` to `mesh.f[:,1]`).  
                          `mesh.f[:,3]` is either the element opposed to `mesh.f[:,3]`,  
                          or the negative of the boundary number shifted by 1. Note that  
                          all boundary edges are last for convenience.

```
>> mesh.f
[[ 7  0  0  4]
 [ 0  8  0  6]
 [ 8  7  0  2]
 [ 5  7  1  3]
 [ 7  1  1  2]
 [ 8  1  2  8]
 [ 6  7  3  4]
 [ 4  8  5  7]
 [ 8  3  5  6]
 [ 2  8  7  8]
 [ 1  5  1 -1]
 [ 5  6  3 -1]
 [ 6  0  4 -1]
 [ 3  4  5 -1]
 [ 0  3  6 -1]
 [ 4  2  7 -1]
 [ 2  1  8 -1]]
```

For this example there is only one geometric boundary `ib = 0` and the corresponding entry in `mesh.f[:,3]` is therefore `-(ib + 1) = -1`

`mesh.t2f (nt,3):`      Triangle to face connectivity. `mesh.t2f[it,in]` contains the face number **plus 1** in element `it` which is opposite node `in` of element `it`. If the face orientation matches the element counterclockwise orientation then the face number is stored, otherwise the negative **minus 1** of the face number is stored.

```
>> mesh.t2f
[[ 1  2  3]
 [11  4  5]
 [-3  6 -5]
 [-4 12  7]
 [13 -1 -7]
 [14  8  9]
 [-2 15 -9]
 [-8 16 10]
 [-6 -10 17]]
```

- **Geometry information**

**mesh.fcurved (nf):** Logical flag indicating which faces are curved. This flag should also be active for straight faces with non-constant Jacobian.

```
>> mesh.fcurved
[False False False False False False False False False True True
 True True True True True]
```

**mesh.tcurved (nt):** Logical flag indicating which triangles have at least a curved face. This flag should also be active for non curved triangles with non-constant Jacobian.

```
>> mesh.tcurved
[False True False True True True True True True]
```

- **Master Element information**

`mesh.porder:` Order of the complete polynomial used for approximation inside each element.

```
>> mesh.porder
3
```

`mesh.plocal (npl,3):` Parametric coordinates of the nodes in the master element. Note that `mesh.plocal[:,0] = 1-mesh.plocal[:,1]-mesh.plocal[:,2]`. Also, `npl = (mesh.porder+1)*(mesh.porder+2)/2`. The order of the nodes is that shown in figure 3.

```
>> mesh.plocal
[[1.00000000e+00 0.00000000e+00 0.00000000e+00]
 [6.66666667e-01 3.33333333e-01 1.85037171e-17]
 [3.33333333e-01 6.66666667e-01 3.70074342e-17]
 [0.00000000e+00 1.00000000e+00 0.00000000e+00]
 [6.66666667e-01 3.70074342e-17 3.33333333e-01]
 [3.33333333e-01 3.33333333e-01 3.33333333e-01]
 [1.85037171e-17 6.66666667e-01 3.33333333e-01]
 [3.33333333e-01 1.85037171e-17 6.66666667e-01]
 [3.70074342e-17 3.33333333e-01 6.66666667e-01]
 [0.00000000e+00 0.00000000e+00 1.00000000e+00]]
```

`mesh.tlocal (ntl,3):` Element vertices for local auxiliary mesh. The element ordering is arbitrary. (Used for refinement and plotting).

```
>> mesh.tlocal
[[0 1 4]
 [1 2 5]
 [2 3 6]
 [1 5 4]
 [2 6 5]
 [4 5 7]
 [5 6 8]
 [5 8 7]
 [7 8 9]]
```

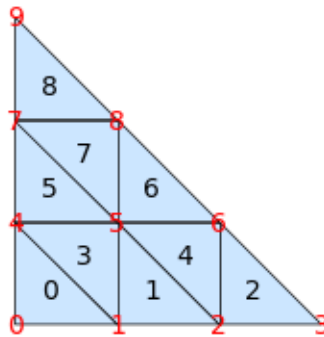


Figure 3: Node positions in master element and local auxiliary mesh connectivity.

- **FEM node locations**

`mesh.dgnodes (npl,2,nt):` `mesh.dgnodes[ipl,[0,1],it]` are the x and y coordinates of the `ipl` local node in element `it`. Note that the nodes that lie on a curved boundary must be placed on the actual geometry as shown in figure 4.

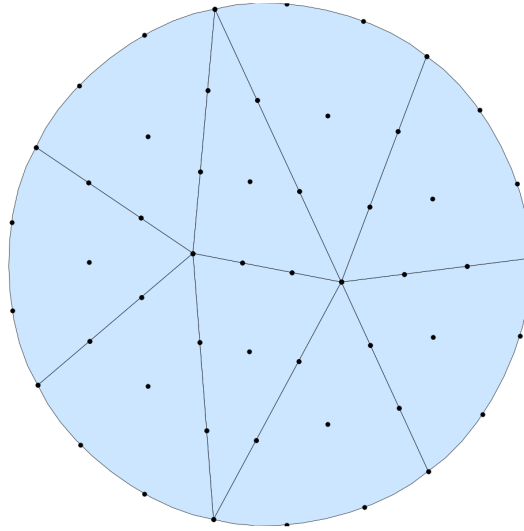


Figure 4: DG node placement for curved geometries.

## 1.2 The master data structure

The `master` data structure contains master element pre-computed information such as the parameters required for numerical integration, the value of the shape functions and their derivatives as well as connectivity information required for efficient assembly.

- **Master Element information**

The variable `master.porder = mesh.porder` and the array `master.plocal = mesh.plocal` are duplicated for convenience.

<code>master.porder:</code>	Order of the complete polynomial used for approximation inside each element.
<code>master.plocal (npl,3):</code>	Parametric coordinates of the nodes in the master element.
<code>master.ploc1d (npl1d,3):</code>	Parametric coordinates for the 1D master element. Here, <code>npl1d = master.porder+1</code>

```
>> master.ploc1d
[[1.  0.  ]
 [0.66666667 0.33333333]
 [0.33333333 0.66666667]
 [0.  1.  ]]
```

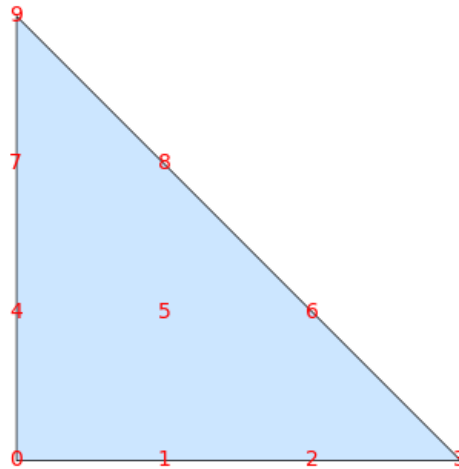


Figure 5: Local node numbering for a cubic triangular element.

- **Node Templates**

<pre> master.corner (3): &gt;&gt; master.corner [0 3 9] </pre>	Node numbers of the three corner nodes (see figure 5).
<pre> master.perm (npl1d,3,2): </pre>	Nodes numbers of the nodes on each edge. Note that, <code>npl1d = master.porder+1</code> . <code>master.perm[:,ifl,0]</code> is the list of nodes on face <code>ifl</code> taken counterclockwise and <code>master.perm[:,ifl,1]</code> is the list of nodes on face <code>ifl</code> taken clockwise (see figure 5).
<pre> &gt;&gt; master.perm[:,:,0] [[3 9 0]  [6 7 1]  [8 4 2]  [9 0 3]] </pre>	
<pre> &gt;&gt; master.perm[:,:,1] [[9 0 3]  [8 4 2]  [6 7 1]  [3 9 0]] </pre>	

- **Numerical Integration**

<pre> master.gpts (ng,2): </pre>	Parametric coordinates $\{(\xi, \eta) \mid \xi \geq 0, \eta \geq 0, 1 - \xi - \eta \geq 0\}$ of the integration points in 2d. Here, <code>ng</code> is the number of 2D integration points.
<pre> master.gwgh (ng): </pre>	Integration weights in 2D.
<pre> master.gp1d (ng1d): </pre>	Parametric coordinates $\{\xi \mid \xi \geq 0, \xi \leq 1\}$ of the integration points in 1d. Here, <code>ng1d</code> is the number of 1D integration points.
<pre> master.gw1d (ng1d): </pre>	Integration weights in 1D.

- **Shape Functions**

<pre> master.shap (npl,3,ng): </pre>	Value of the 2D cardinal (or nodal) shape functions and their derivatives evaluated at the integration points. Here, <code>master.shap[:,0,:]</code> contains the value of the shape function and <code>master.shap[:,[1,2],:]</code> contains the values of the derivatives of the shape functions with respect to $\xi$ and $\eta$ .
<pre> master.sh1d (npl1d,2,ng1d): </pre>	Value of the 1D cardinal (or nodal) shape functions and their derivatives evaluated at the integration points. Here, <code>master.sh1d[:,0,:]</code> contains the value of the shape function and <code>master.sh1d[:,1,:]</code> contains the values of the derivatives of the shape functions with respect to $\xi$ .



- **Pre-computed Element Matrices**

<code>master.mass[npl,npl]:</code>	Mass matrix in 2D. The entry <code>master.mass[i,j]</code> corresponds to $\int_{K'} \phi_i \phi_j dK'$ , where $K'$ is the master element
<code>master.conv[npl,npl,2]:</code>	Convection matrices in 2D. The entry <code>master.conv[i,j,0]</code> corresponds to $\int_{K'} \phi_i \phi_{j,\xi} dK'$ , whereas <code>master.conv[i,j,1]</code> corresponds to $\int_{K'} \phi_i \phi_{j,\eta} dK'$ .
<code>master.ma1d[npl1d,npl1d]:</code>	Mass matrix in 1D. The entry <code>master.ma1d[i,j]</code> corresponds to $\int_{K'_{1d}} \phi_i \phi_j dK'_{1d}$ , where $K'_{1d}$ is the 1D master element.

### 1.3 The app data structure

The `app` data structure contains information regarding the application. In particular, it specifies the equations to be solved by providing pointers to the flux functions, but it also contains control parameters as well as the data and the arguments necessary to evaluate the equation fluxes. The `app` data structure can be assembled by hand for simple applications or created by a `mkapp` function.

- **Control parameters**

<code>app.nc</code> :	Number of components in the vector of unknowns. For scalar equations <code>app.nc = 1</code> .
<code>app.pg</code> :	Logical flag which must be set to <code>true</code> when it is necessary for the flux functions to have access to the <code>x</code> and <code>y</code> coordinates in order to evaluate the fluxes. Setting it to <code>false</code> avoids costly interpolation and this should be the choice when the fluxes are independent of the spatial coordinates.
<code>app.arg</code> :	Arguments to be passed to the flux functions <code>app.finv</code> , <code>app.finvb</code> and <code>app.finvv</code> below. <code>app.arg</code> is a cell array and can contain arrays, pointers to functions, etc. in its entries.

- **Boundary Conditions**

<code>app.bcm[ngb]</code> :	Boundary conditions assignment. For each of the <code>ngb</code> geometric boundaries <code>igb</code> , <code>app.bcm[igb]</code> assigns the appropriate boundary condition. For instance, if we are solving an equation on a square domain, we will typically have four geometric boundaries ( <code>ngb = 4</code> ), but we may only have two types of boundary conditions. In this case, <code>app.bcm</code> may look like, <code>app.bcm = [0,0,1,0]</code> . Note that the geometric boundary to which a given mesh boundary edge <code>if</code> belongs is given by <code>-mesh.f[if,3] - 1</code> .
<code>app.bcs[nbt,app.nc]</code> :	Far-field states for boundary condition application. The total number of boundary types is given by <code>nbt</code> and this number has to be larger or equal to the maximum in <code>app.bcm</code> . For instance, if <code>app.bcm = [0,0,1,0]</code> , <code>nbt</code> must be at least 2, and <code>app.bcs[nbt,app.nc]</code> contains two far field states that maybe used in order to determine the boundary conditions, of types 0 and 1.

- **Function Pointers**

In order to describe the different flux functions, we write our system of conservation laws in the form

$$\frac{\partial u}{\partial t} + \nabla \cdot [\mathbf{F}^{\text{inv}}(u, \mathbf{x}, t)] = \mathbf{S}(u, \mathbf{q}, \mathbf{x}, t),$$

The DG variational form the first equation is

$$\begin{aligned}
& \sum_{K \in \mathcal{T}_h} \int_K \frac{\partial u_h}{\partial t} w \, d\mathbf{x} \\
& + \underbrace{\sum_{e \in \mathcal{E}_{ih}} \int_e \hat{\mathbf{F}}^{\text{inv}}(u_h^+, u_h^-, \mathbf{x}, t) \cdot [w] \, ds}_{\text{inter-element term I}} + \underbrace{\sum_{e \in \partial\Omega} \int_e \hat{\mathbf{F}}^{\text{inv}}(u_h^+, u_b, \mathbf{x}, t) \cdot [w] \, ds}_{\text{boundary term I}} - \underbrace{\sum_{K \in \mathcal{T}_h} \int_K \mathbf{F}^{\text{inv}}(u_h, \mathbf{x}, t) \cdot \nabla w \, d\mathbf{x}}_{\text{volume term I}} = \\
& + \underbrace{\sum_{e \in \mathcal{E}_{ih}} \int_e \hat{\mathbf{F}}^{\text{vis}}(u_h^+, u_h^-, \mathbf{q}_h^+, \mathbf{q}_h^-, \mathbf{x}, t) \cdot [w] \, ds}_{\text{inter-element term II}} + \underbrace{\sum_{e \in \partial\Omega} \int_e \hat{\mathbf{F}}^{\text{vis}}(u_h^+, \mathbf{q}_h^+, u_b, \mathbf{q}_b, \mathbf{x}, t) \cdot [w] \, ds}_{\text{boundary term II}} - \underbrace{\sum_{K \in \mathcal{T}_h} \int_K \mathbf{F}^{\text{vis}}(u_h, \mathbf{q}_h, \mathbf{x}, t) \cdot \nabla w \, d\mathbf{x}}_{\text{volume term II}} = \\
& \underbrace{\sum_{K \in \mathcal{T}_h} \int_K \mathcal{S}(u_h, \mathbf{q}_h, \mathbf{x}, t) w \, d\mathbf{x}}_{\text{source term}}
\end{aligned}$$

whereas for the second equation we have

$$\sum_{K \in \mathcal{T}_h} \int_K \mathbf{q}_h \cdot \mathbf{v} \, d\mathbf{x} = \sum_{e \in \mathcal{E}_{ih}} \int_e \hat{u}_h(u_h^+, u_h^-) [\mathbf{v}] \, ds + \underbrace{\sum_{e \in \partial\Omega} \int_e \hat{u}_h(u_h^+, u_b) [\mathbf{v}] \, ds}_{\text{boundary term III}} - \sum_{K \in \mathcal{T}_h} \int_K u_h \nabla \cdot \mathbf{v} \, d\mathbf{x}$$

Here, we are already assuming that the numerical flux  $\hat{u}$  is not a function of  $\mathbf{q}_h$  (that is  $\mathcal{C}_{22} = 0$ ).

We will then construct functions that evaluate the fluxes at the Gauss points. Note that these functions evaluate fluxes at Gauss points and are independent of the finite element technology or algorithm i.e. interpolation, integration rules etc.

The functions are:

<code>app.finv:</code>	Pointer to function for the evaluation of the inviscid interface flux $\hat{\mathbf{F}}^{\text{inv}}(u_h^+, u_h^-, \mathbf{x}, t)$ (inter-element term I) <code>fn = app.finv( up, um, np, p, app.arg, time)</code>
<code>app.finvb:</code>	Pointer to function for the evaluation of the inviscid boundary flux $\hat{\mathbf{F}}^{\text{inv}}(u_h^+, u_b, \mathbf{x}, t)$ (boundary term I) <code>fn = app.finvb( up, np, ibt, uinf, p, app.arg, time)</code>
<code>app.finvv:</code>	Pointer to function for the evaluation of the inviscid volume flux $\hat{\mathbf{F}}^{\text{inv}}(u_h, \mathbf{x}, t)$ (volume term I) <code>[fx,fy] = app.finvv( u, p, app.arg, time)</code>
<code>app.fvisi:</code>	Pointer to function for the evaluation of the viscous interface flux $\hat{\mathbf{F}}^{\text{vis}}(u_h^+, u_h^-, \mathbf{q}_h^+, \mathbf{q}_h^-, \mathbf{x}, t)$ (inter-element term II) <code>fn = app.fvisi( up, um, qp, qm, np, p, app.arg, time)</code>
<code>app.fvisb:</code>	Pointer to function for the evaluation of the viscous boundary flux $\hat{\mathbf{F}}^{\text{vis}}(u_h^+, \mathbf{q}_h^+, u_b, \mathbf{q}_b, \mathbf{x}, t)$ (boundary term II) <code>fn = app.fvisb( up, qp, np, ibt, uinf, p, app.arg, time)</code>
<code>app.fvisv:</code>	Pointer to function for the evaluation of the viscous volume flux $\hat{\mathbf{F}}^{\text{vis}}(u_h, \mathbf{q}_h, \mathbf{x}, t)$ (volume term II) <code>[fx,fy] = app.fvisv( u, q, p, app.arg, time)</code>
<code>app.fvisubv:</code>	Pointer to function for the evaluation of the boundary flux for $u$ $\hat{u}(u_h^+, u_b, \mathbf{x}, t)$ (boundary term III) <code>ub = app.fvisub( up, ibt, uinf, p, app.arg, time)</code>
<code>app.src:</code>	Pointer to function for the evaluation of the source term $\mathcal{S}(u_h, \mathbf{q}_h, \mathbf{x}, t)$ (source term) <code>sr = app.src( u, q, p, app.arg, time)</code>