Stéphane Clain

FVLib

The Finite Volume Library Reference Manual

Part I

The basic layer

Index for the basic layer

- *mesh file format*
- *vector file format*
- *parameter file format*
- *table file format*

- FVCell1D, FVCell2D, FVCell3D,
- FVEdge2D , FVEdge3D,
- FVFace3D,
- FVMesh1D, FVMesh2D, FVMesh3D,
- FVStencil, FVRecons1D, FVRecons2D, FVRecons3D,
- FVVertex1D, FVVertex2D, FVVertex3D
- GMElement, Gmsh,

- FVDenseM, FVSparseM, FVVect, FVSparseV, FVVKrylov

- FVGaussPoint1D, FVGaussPoint2D, FVGaussPoint3D
- FVio
- FVPoint1D, FVPoint2D, FVPoint3D, FVPoint4D
- Parameter, Table

Chapter 1

File format

1.1 The mesh file format

Short description 1

The **FVlib** *mesh file format.* *Back to index*

1D mesh.

Listing 1.1: One dimensional mesh file format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FVLIB>
    <MESH dim="1"    name="noname">
        <VERTEX nbvertex="11">
   <!--label  code   coord   -->
        1      1     0.0000
        2      0     0.1000
        3      0     0.2000
        4      0     0.3000
        5      0     0.4000
        6      0     0.5000
        7      0     0.6000
        8      0     0.7000
        9      0     0.8000
        10     0     0.9000
        11     2     1.0000
        </VERTEX>
        <CELL nbcell="10">
<!--label  code  nbVert list of vertices   -->
        1    6      2     1    2
        2    19     2     2    3
        3    8      2     3    4
        4    0      2     4    5
        5    0      2     5    6
        6    0      2     6    7
        7    0      2     7    8
        8    0      2     8    9
        9    5      2     9    10
        10   6      2     10   11
        </CELL>
    </MESH>
</FVLIB>
```

2D mesh.

Listing 1.2: Two dimensional mesh file format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FVLIB>
    <MESH dim="2"    name="mesh2D">
```

```
        <VERTEX nbvertex="17">
   <!--label  code   coord   -->
        1      1     0.0000   1.0000
        2      1     0.8000   1.0000
        3      1     1.0000   1.0000
        4      0     1.0000   0.8000
        5      2     1.0000   0.0000
        6      2     0.8000   0.0000
        7      2     0.0000   0.0000
        8      0     0.0000   0.5000
        9      0     0.2000   0.8000
        10     0     0.8000   0.8000
        11     0     0.8000   0.2000
        12     0     0.5000   0.2000
        13     0     0.2000   0.2000
        14     0     0.2000   0.4000
        15     0     0.3000   0.5000
        16     0     0.5000   0.5000
        17     0     0.5000   0.8000
        </VERTEX>
        <EDGE nbedge="26">
<!-- label  code nbvert list of vertices  -->
        1    0      2     1    2
        2    0      2     2    3
        3    0      2     3    4
        4    0      2     4    5
        5    0      2     5    6
        6    0      2     6    7
        7    0      2     7    8
        8    0      2     8    1
        9    0      2     1    9
        10   0      2     2    10
        11   0      2     10   4
        12   0      2     6    11
        13   0      2     7    13
        14   0      2     8    14
        15   0      2     8    9
        16   0      2     9    17
        17   0      2     17   10
        18   0      2     11   10
        19   0      2     11   12
        20   0      2     12   13
        21   0      2     13   14
        22   0      2     14   15
        23   0      2     9    15
        24   0      2     15   16
        25   0      2     16   17
        26   5      2     16   12
        </EDGE>
        <CELL nbcell="10">
<!--label  code  nbEdge list of edges   -->
        1    0      5     1    10   9    16   17
        2    0      4     2    3    11   10
        3    0      5     4    5    11   12   18
        4    0      5     6    20   13   12   19
        5    0      4     7    21   13   14
        6    0      3     15   8    9
        7    0      4     22   23   15   14
        8    0      4     23   16   24   25
        9    0      5     17   18   19   25   26
        10   4      5     20   21   22   24   26
        </CELL>
    </MESH>
</FVLIB>
```

3D mesh.

Listing 1.3: Three dimensional mesh file format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FVLIB>
    <MESH dim="3"     name="mesh3D">
        <VERTEX nbvertex="16">
   <!--label  code   coord   -->
        1      1      0.0000   0.0000   0.0000
        2      1      0.0000   0.0000   1.0000
        3      1      1.0000   0.0000   1.0000
        4      0      1.0000   0.0000   0.0000
        5      2      0.0000   1.0000   0.0000
        6      2      0.0000   1.0000   1.0000
        7      2      1.0000   1.0000   1.0000
        8      0      1.0000   1.0000   0.0000
        9      0      0.3000   0.3000   0.3000
        10     0      0.3000   0.3000   0.7000
        11     0      0.7000   0.3000   0.7000
        12     0      0.7000   0.3000   0.3000
        13     0      0.3000   0.7000   0.3000
        14     0      0.3000   0.7000   0.7000
        15     0      0.7000   0.7000   0.7000
        16     0      0.7000   0.7000   0.3000
        </VERTEX>
        <EDGE nbedge="32">
<!-- label  code nbvert list of vertices  -->
        1    0      2     1    2
        2    0      2     2    3
        3    0      2     3    4
        4    0      2     4    1
        5    0      2     5    6
        6    0      2     6    7
        7    0      2     7    8
        8    0      2     8    5
        9    0      2     1    5
        10   0      2     2    6
        11   0      2     3    7
        12   0      2     4    8
        13   0      2     9    10
        14   0      2     10   11
        15   0      2     11   12
        16   0      2     12   9
        17   0      2     13   14
        18   0      2     14   15
        19   0      2     15   16
        20   0      2     16   13
        21   0      2     9    13
        22   0      2     10   14
        23   0      2     11   15
        24   0      2     12   16
        25   0      2     1    9
        26   5      2     2    10
        27   0      2     3    11
        28   0      2     4    12
        29   0      2     5    13
        30   0      2     6    14
        31   0      2     7    15
        32   5      2     8    16
        </EDGE>
        <FACE nbface="24">
<!-- label  code nbEdge list of edges  -->
        1    0      4     1    2    3    4
        2    0      4     1    9    5    10
        3    0      4     5    6    7    8
        4    0      4     3    11   7    12
        5    0      4     2    10   6    11
        6    0      4     9    8    12   4
        7    0      4     13   14   15   16
        8    0      4     21   13   22   17
        9    0      4     17   18   19   20
        10   0      4     15   19   23   24
        11   0      4     14   18   22   23
        12   0      4     16   20   21   24
        13   0      4     2    14   26   27
        14   0      4     10   22   26   30
        15   0      4     30   31   6    18
        16   0      4     11   31   27   23
        17   0      4     4    16   25   28
        18   0      4     9    21   25   29
        19   0      4     8    20   29   32
        20   0      4     12   24   28   32
        21   0      4     1    13   25   26
        22   0      4     5    17   29   30
        23   0      4     7    19   31   32
        24   0      4     3    15   27   28
        </FACE>
        <CELL nbcell="7">
<!--label  code  nbFace list of faces   -->
        1    0      6     1    7    13   17   24   21
        2    0      6     2    8    14   18   21   22
        3    0      6     3    9    15   19   22   23
        4    0      6     4    10   16   20   23   24
        5    0      6     5    11   13   14   15   16
        6    0      6     6    12   17   18   19   20
        7    0      6     7    8    9    10   11   12
        </CELL>
    </MESH>
</FVLIB>
```

Example.

1.2 The vector file format

Short description 2

The **FVlib** *vector file format.* *Back to index*

Listing 1.4: vector file format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FVLIB>
    <FIELD size="10"  nbvec="1"  time="0"  name="noname">
  1.200000000000e+00  1.200000000000e+00  1.200000000000e+00  1.200000000000e+00
  1.200000000000e+00  1.200000000000e+00  1.200000000000e+00  1.200000000000e+00
  1.200000000000e+00  1.200000000000e+00
    </FIELD>
    <FIELD size="10"  nbvec="2"  time="3.000000000000e-01"  name="noname">
  3.000000000000e-13 3.100000000000e+00  3.000000000000e-13 3.100000000000e+00
  3.000000000000e-13 3.100000000000e+00  3.000000000000e-13 3.100000000000e+00
  3.000000000000e-13 3.100000000000e+00  3.000000000000e-13 3.100000000000e+00
  3.000000000000e-13 3.100000000000e+00  3.000000000000e-13 3.100000000000e+00
  3.000000000000e-13 3.100000000000e+00  3.000000000000e-13 3.100000000000e+00
    </FIELD>
    <FIELD size="10"  nbvec="3"  time="5.000000000000e-01"  name="density">
  1.234567890120e+04 3.100000000000e+00 0.000000000000e+00 1.234567890120e+04
  3.100000000000e+00 0.000000000000e+00 1.234567890120e+04 3.100000000000e+00
  0.000000000000e+00 1.234567890120e+04 3.100000000000e+00 0.000000000000e+00
  1.234567890120e+04 3.100000000000e+00 0.000000000000e+00 1.234567890120e+04
  3.100000000000e+00 0.000000000000e+00 1.234567890120e+04 3.100000000000e+00
  0.000000000000e+00 1.234567890120e+04 3.100000000000e+00 0.000000000000e+00
  1.234567890120e+04 3.100000000000e+00 0.000000000000e+00 1.234567890120e+04
  3.100000000000e+00 0.000000000000e+00
    </FIELD>
</FVLIB>
```

Example.

1.3 The parameter file format

Short description 3

The **FVlib** *parameter file format.*

Back to index

Listing 1.5: parameter file format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- File Parameter -->
<!-- december 2010 -->
<FVLIB>
    <PARAMETER >
           <parameter name="toto" distance="12.13" code="25" result="-1"/>
            <parameter name2="tutu" radius="12.13" after="12" before="-1"/>
            <parameter name3="titi" />
           <parameter name4="toto" distance4="12.13" code4="12" result4="-1"/>
           <parameter name5="toto" distance5="12.13" code5="12" result5="-1"/>
    </PARAMETER>
</FVLIB>
```

Example.

1.4 The table file format

Short description 4

The **FVlib** *table file format.* *Back to index*

Listing 1.6: table file format

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- File table.xml -->
<!-- december 2010 -->
<FVLIB>
    <TABLE label="air">
          <FUNCTION label="P">
                 <VARIABLE label="rho" nb_pts="5" min="0" max="1"/>
                 <VARIABLE label="e" nb_pts="4" min="10" max="12"/>
                 <DATA>
           1.0   2.0   5.0   7.0
           2.0   3.0   5.0   8.0
           7.0   2.0   5.0   9.0
           0.0   2.0   8.0  10.0
          11.0  20.0  25.0  30.0
                 </DATA>
          </FUNCTION>
           <FUNCTION label="c">
                 <VARIABLE label="rho" nb_pts="5" min="0" max="1"/>
                 <VARIABLE label="P" nb_pts="4" min="0" max="1"/>
                 <DATA>
           1.0   2.0   5.0   7.0
           2.0   3.0   5.0   8.0
           7.0   2.0   5.0   9.0
           0.0   2.0   8.0  10.0
          11.0  20.0  25.0  30.0
                 </DATA>
          </FUNCTION>
           <FUNCTION label="CP">
                 <VARIABLE label="T" nb_pts="5" min="0" max="1"/>
                 <DATA>
           1.0   2.0   5.0   7.0  2.0
                 </DATA>
          </FUNCTION>
           <FUNCTION label="freq">
                 <VARIABLE label="t" nb_pts="5" min="0" max="1"/>
                 <VARIABLE label="p" nb_pts="4" min="10" max="12"/>
                 <VARIABLE label="T" nb_pts="2" min="10" max="12"/>
                 <DATA>
           1.0   2.0   5.0   7.0
           2.0   3.0   5.0   8.0
           7.0   2.0   5.0   9.0
           0.0   2.0   8.0  10.0
          11.0  20.0  25.0  30.0
           1.0   2.0   5.0   7.0
           2.0   3.0   5.0   8.0
           7.0   2.0   5.0   9.0
           0.0   2.0   8.0  10.0
          11.0  20.0  25.0  30.0
                 </DATA>
          </FUNCTION>
    </TABLE>
</FVLIB>
```

Example.

Chapter 2

Mesh

2.1 FVCell1D

`FVCell1D` Class Reference

Short description 5

Class for the one-dimensional cell. *Back to index*

Field.

`size_t label`

The label of the cell.

`size_t code`

The code of the cell.

`double length`

The length of the cell.

`FVPoint1D<double> centroid`

The x-coordinate of the cell centroid.

`FVPoint1D<double> first_normal,second_normal`

Outward normal direction at the first and second interface

`FVVertex1D *firstVertex,*secondVertex`

Pointer to the first and second vertex of the cell.

Method.

`FVCell1D()`

Constructor method.

`~FVCell1D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVCell1D`. The value is the length of the cell.

`double getMeanValue(double (&f))`

Return a sixth-order approximation of the mean value of funtion `f` on the cell. The function must be declared as `double f(FVPoint1D<double> )`.

`double getMeanValue(double (&f), Parameter &para)`

Return a sixth-order approximation of the mean value of funtion `f` on the cell. The function must be declared as `double f(FVPoint1D<double>, Parameter &para)`.

`FVVertex1D * beginVertex()`

Initialize the internal pointer to the first vertex of the cell and return the address.

`FVVertex1D * nextVertex()`

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

```
size_t getLocalIndexVertex()
```

When using the `nextVertex` method, return the local index of the current vertex.

```
void setCode2Vertex(size_t val=0)
```

Set `val` (0 default value) to the two vertices code. ***Function.***

```
inline bool isEqual(FVCell1D *c1, FVCell1D*c2)
```

Return `true` if `c1` and `c2` have the same vertices.

Example.

2.2 FVCell2D

FVCell2D Class Reference

Short description 6
Class for the two-dimensional cell.

Back to index

Field.

`size_t label`

The label of the cell.

`size_t code`

The code of the cell.

`double perimeter`

The perimeter of the cell.

`double area`

The area of the cell.

`size_t nb_vertex`

The number of vertices of the cell.

`size_t nb_edge`

The number of edges of the cell.

`FVPoint2D<double> centroid`

The (x, y)-coordinates of the cell centroid.

`FVVertex2D * vertex[NB_VERTEX_PER_CELL_2D]`

Pointer table to the vertices of the cell from 0 to `nb_vertex`-1. NB_VERTEX_PER_CELL_2D is the maximum number of vertices in a cell defined in file `FVLib_config.h`.

`FVEdge2D * edge[NB_EDGE_PER_CELL_2D]`

Pointer table to the edges of the cell from 0 to `nb_edge`-1. NB_EDGE_PER_CELL_2D is the maximum number of edges in a cell defined in file `FVLib_config.h`.

`FVPoint2D<double> cell2edge[NB_EDGE_PER_CELL_2D]`

Table of the cell–centroid edge–centroid vector of the cell from 0 to `nb_edge`-1. NB_EDGE_PER_CELL_2D is the maximum number of edges in a cell defined in file `FVLib_config.h`.

Method.

`FVCell2D()`

Constructor method.

`~FVCell2D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVCell2D`. The value is the area of the cell.

`double getMeanValue(double (&f))`

Return a sixth-order approximation of the mean value of funtion `f` on the cell. The function must be declared as `double f(FVPoint2D<double> )`.

`double getMeanValue(double (&f), Parameter &para)`

Return a sixth-order approximation of the mean value of funtion `f` on the cell. The function must be declared as `double f(FVPoint2D<double>, Parameter &para)`.

`FVVertex2D * beginVertex()`

Initialize the internal pointer to the first vertex of the cell and return the address.

`FVVertex2D * nextVertex()`

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

`size_t getLocalIndexVertex()`

When using the `nextVertex` method, return the local index of the current vertex.

`FVEdge2D * beginEdge()`

Initialize the internal pointer to the first edge of the cell and return the address.

`FVEdge2D * nextEdge()`

Return the pointer of the current edge and move to the next one. If we reach the end of the list, return `NULL`.

`size_t getLocalIndexEdge()`

When using the `nextEdge` method, return the local index of the current vertex.

`FVPoint2D<double> getCell2Edge()`

Return the cell-centroid to edge-centroid vector of the current edge handled by `beginEdge` and `nextEdge`.

`setCode2Vertex(size_t val=0)`

Set `val` (0 default value) to the code of the vertices.

`setCode2Edge(size_t val=0)`

Set `val` (0 default value) to the code of the edges.

Function.

`inline bool isEqual(FVCell2D *c1, FVCell2D*c2)`

Return `true` if `c1` and `c2` have the same edges.

Example.

Listing 2.1: Loop over the edges of a 2D cell

```
FVEdge2D *ptr_e;
```

```
FVCell2D c;
for(c.beginEdge();(ptr_e=c.nextEdge());)
    {
    cout<<"label of edge "<<ptr_e->label<<endl;
    cout<<"vector cell-centroid edge-centroid"<<c.getCell2Edge()<<endl;
    }
```

Listing 2.2: Loop over the vertices of a 2D cell

```
FVVertex2D *ptr_v;
FVCell2D c;
for(c.beginVertex();(ptr_v=c.nextVertex());)
    {
    cout<<"label of vertex "<<ptr_v->label<<endl;
    cout<<"coordinate of the vertex "<<ptr_v->coord<<endl;
    }
```

2.3 FVCell3D

FVCell3D Class Reference

Short description 7

Class for the three-dimensional cell. *Back to index*

Field.

`size_t label`

The label of the cell.

`size_t code`

The code of the cell.

`double surface`

The surface of the cell.

`double volume`

The volume of the cell.

`size_t nb_vertex`

The number of vertices of the cell.

`size_t nb_face`

The number of faces of the cell.

`FVPoint3D<double> centroid`

The (x, y, z)-coordinates of the cell centroid.

`FVVertex3D * vertex[NB_VERTEX_PER_CELL_3D]`

Pointer table to the vertices of the cell from 0 to `nb_vertex`-1. NB_VERTEX_PER_CELL_3D is the maximum number of vertices in a cell defined in file `FVLib_config.h`.

`FVFACE3D * face[NB_FACE_PER_CELL_3D]`

Pointer table to the faces of the cell from 0 to `nb_face`-1. NB_FACE_PER_CELL_3D is the maximum number of edges in a cell defined in file `FVLib_config.h`.

`FVPoint3D<double> cell2face[NB_FACE_PER_CELL_3D]`

Table of the cell–centroid face–centroid vector of the cell from 0 to `nb_face`-1. NB_FACE_PER_CELL_3D is the maximum number of edges in a cell defined in file `FVLib_config.h`.

Method.

`FVCell3D()`

Constructor method.

`~FVCell3D()`

Destructor method.

```
double measure()
```

Return the measure of the geometrical entity `FVCell3D`. The value is the volume of the cell.

```
double getMeanValue(double (&f))
```

Return a sixth-order approximation of the mean value of funtion `f` on the cell. The function must be declared as `double f(FVPoint3D<double> )`.

```
double getMeanValue(double (&f), Parameter &para)
```

Return a sixth-order approximation of the mean value of funtion `f` on the cell. The function must be declared as `double f(FVPoint3D<double>, Parameter &para)`.

```
FVVertex3D * beginVertex()
```

Initialize the internal pointer to the first vertex of the cell and return the address.

```
FVVertex3D * nextVertex()
```

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

```
size_t getLocalIndexVertex()
```

When using the `nextVertex` method, return the local index of the current vertex.

```
FVFace3D * beginFace()
```

Initialize the internal pointer to the first face of the cell and return the address.

```
FVFace3D * nextFace()
```

Return the pointer of the current face and move to the next one. If we reach the end of the list, return `NULL`.

```
size_t getLocalIndexFace()
```

When using the `nextFace` method, return the local index of the current vertex.

```
FVPoint3D<double> getCell2Face()
```

Return the cell-centroid to face-centroid vector of the current edge handled by `beginFace` and `nextFace`.

```
setCode2Vertex(size_t val=0)
```

Set `val` (0 default value) to the code of the vertices.

```
setCode2Edge(size_t val=0)
```

Set `val` (0 default value) to the code of the edges.

```
setCode2Face(size_t val=0)
```

Set `val` (0 default value) to the code of the faces.

Function.

```
inline bool isEqual(FVCell3D *c1, FVCell3D*c2)
```

Return `true` if `c1` and `c2` have the same faces.

Example.

Listing 2.3: Loop over the faces of a 3D cell

```
FVFace3D *ptr_f;
FVCell3D c;
for(c.beginFace();(ptr_f=c.nextFace());)
    {
    cout<<"label of face "<<ptr_f->label<<endl;
    cout<<"vector cell-centroid face-centroid"<<c.getCell2Face()<<endl;
    }
```

Listing 2.4: Loop over the vertices of a 3D cell

```
FVVertex2D *ptr_v;
FVCell3D c;
for(c.beginVertex();(ptr_v=c.nextVertex());)
    {
    cout<<"label of vertex "<<ptr_v->label<<endl;
    cout<<"coordinate of the vertex "<<ptr_v->coord<<endl;
    }
```

2.4 FVEdge2D

FVEdge2D Class Reference

Short description 8

Class for the two-dimensional edge.

Back to index

Field.

`size_t label`

The label of the edge.

`size_t code`

The code of the edge.

`double length`

The length of the edge.

`size_t nb_vertex`

The number of vertices of the edge (always 2).

`size_t nb_cell`

The number of cell of the edge (always 2).

`FVPoint2D<double> centroid`

The (x, y)-coordinates of the edge centroid.

`FVVertex2D *firstVertex,*secondVertex`

Pointer to the first and second vertex of the edge.

`FVCell2D *leftCell,*rightCell`

Pointer to the left and right cell of the edge. `leftCell` is always the inner cell. `rightCell=NULL` if the edge is on the boundary.

`FVPoint2D<double> normal`

The (x, y)-coordinate of the edge normal vector from left to right.

Method.

`FVEdge2D()`

Constructor method.

`~FVEdge2D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVEdge2D`. The value is the length of the edge.

```
double getMeanValue(double (&f))
```

Return a sixth-order approximation of the mean value of funtion `f` on the edge. The function must be declared as `double f(FVPoint2D<double> )`.

```
double getMeanValue(double (&f), Parameter &para)
```

Return a sixth-order approximation of the mean value of funtion `f` on the edge. The function must be declared as `double f(FVPoint2D<double>, Parameter &para)`.

```
FVVertex2D * beginVertex()
```

Initialize the internal pointer to the first vertex of the cell and return the address.

```
FVVertex2D * nextVertex()
```

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

```
size_t getLocalIndexVertex()
```

When using the `nextVertex` method, return the local index of the current vertex.

```
setCode2Vertex(size_t val=0)
```

Set `val` (0 default value) to the code of the vertices.

Function.

```
inline bool isEqual(FVEdge2D *c1, FVEdge2D*c2)
```

Return `true` if `c1` and `c2` have the same vertices.

Example.

2.5 FVEdge3D

FVEdge3D Class Reference

Short description 9

Class for the three-dimensional edge. *Back to index*

Field.

`size_t label`

The label of the edge.

`size_t code`

The code of the edge.

`double length`

The length of the edge.

`size_t nb_vertex`

The number of vertices of the edge (always 2).

`size_t nb_cell`

The number of cell of the edge (always 2).

`FVPoint3D<double> centroid`

The (x, y, z)-coordinates of the edge centroid.

`FVVertex3D *firstVertex,*secondVertex`

Pointer to the first and second vertex of the edge.

Method.

`FVCell3D()`

Constructor method.

`~FVCell3D()`

Destructor method.

`FVVertex3D * beginVertex()`

Initialize the internal pointer to the first vertex of the cell and return the address.

`FVVertex3D * nextVertex()`

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

`size_t getLocalIndexVertex()`

When using the `nextVertex` method, return the local index of the current vertex.

`double measure()`

Return the measure of the geometrical entity `FVEdge3D`. The value is the length of the edge.

```
double getMeanValue(double (&f))
```

Return a sixth-order approximation of the mean value of funtion `f` on the edge. The function must be declared as `double f(FVPoint3D<double> )`.

```
double getMeanValue(double (&f), Parameter &para)
```

Return a sixth-order approximation of the mean value of funtion `f` on the edge. The function must be declared as `double f(FVPoint3D<double>, Parameter &para)`.

```
setCode2Vertex(size_t val=0)
```

Set `val` (0 default value) to the code of the vertices.

Function.

```
inline bool isEqual(FVEdge3D *c1, FVEdge3D*c2)
```

Return `true` if `c1` and `c2` have the same vertices.

Example.

2.6 FVFace3D

FVFace3D Class Reference

Short description 10

Class for the three-dimensional face.

Back to index

Field.

`size_t label`

The label of the face.

`size_t code`

The code of the face.

`double perimeter`

The perimeter of the face.

`double area`

The area of the face.

`size_t nb_vertex`

The number of vertices of the edge (always greater than 3).

`size_t nb_cell`

The number of cell of the edge (always greater than 3).

`FVPoint3D<double> centroid`

The (x, y, z)-coordinate of the face centroid.

`FVCell3D *leftCell,*rightCell`

Pointer to the left and right cell of the edge. `leftCell` is always the inner cell. `rightCell`=NULL if the edge is on the boundary.

`FVVertex3D *vertex[NB_VERTEX_PER_FACE_3D]`

Pointer table to the vertices of the face from 0 to `nb_vertex`-1. NB_VERTEX_PER_FACE_3D is the maximum number of vertices in a face defined in file `FVLib_config.h`.

`FVVEdge3D *edge[NB_EDGE_PER_FACE_3D]`

Pointer table to the edges of the face from 0 to `nb_edge`-1. NB_EDGE_PER_FACE_3D is the maximum number of edges in a face defined in file `FVLib_config.h`.

`FVPoint3D<double> normal[NB_EDGE_PER_FACE_3D]`

The (x, y, z)-coordinates of the normal vector from left to right of a subtriangle i defined by the `edge`[i] and the centroid of the face with i from 0 to `nb_edge`-1..

Method.

`FVFace3D()`

Constructor method.

`~FVFace3D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVFace3D`. The value is the area of the face.

`double getMeanValue(double (&f))`

Return a sixth-order approximation of the mean value of funtion `f` on the face. The function must be declared as `double f(FVPoint3D<double> )`.

`double getMeanValue(double (&f), Parameter &para)`

Return a sixth-order approximation of the mean value of funtion `f` on the face. The function must be declared as `double f(FVPoint3D<double>, Parameter &para)`.

`FVVertex3D * beginVertex()`

Initialize the internal pointer to the first vertex of the face and return the address.

`FVVertex3D * nextVertex()`

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

`size_t getLocalIndexVertex()`

When using the `nextVertex` method, return the local index of the current vertex.

`FVEdge3D * beginEdge()`

Initialize the internal pointer to the first edge of the face and return the address.

`FVEdge3D * nextEdge()`

Return the pointer of the current edge and move to the next one. If we reach the end of the list, return `NULL`.

`size_t getLocalIndexEdge()`

When using the `nextEdge` method, return the local index of the current vertex.

`FVPoint3D<double> getNormal()`

Return the outward normal vector of the subtriangle defined by the face centroid and the current edge handled by `beginEdge` and `nextEdge`.

`setCode2Vertex(size_t val=0)`

Set `val` (0 default value) to the code of the vertices.

`setCode2Edge(size_t val=0)`

Set `val` (0 default value) to the code of the edges.

Function.

`inline bool isEqual(FVFace3D *c1, FVFace3D*c2)`

Return `true` if `c1` and `c2` have the same vertices.

Example.

Listing 2.5: Loop over the edges of a 3D face

```
FVEdge3D *ptr_e;
FVFace3D f;
for(f.beginEdge();(ptr_e=f.nextEdge());)
    {
    cout<<"label of edge "<<ptr_e->label<<endl;
    cout<<"outwer normal vector  face-centroid - edge"<<f.getNormal()<<endl;
    }
```

Listing 2.6: Loop over the vertices of a 3D face

```
FVVertex2D *ptr_v;
FVFace3D f;
for(f.beginVertex();(ptr_v=f.nextVertex());)
    {
    cout<<"label of vertex "<<ptr_v->label<<endl;
    cout<<"coordinate of the vertex "<<ptr_v->coord<<endl;
    }
```

2.7 FVMesh1D

FVMesh1D Class Reference

Short description 11
Class for the one-dimensional mesh. *Back to index*

Field.

All the fields are `private`.

Method.

`FVMesh1D()`

Default Constructor method.

`FVMesh1D(const char * filename)`

Constructor method which load the **FVlib** format mesh in file `filename`.

`void read(const char * filename)`

Load the **FVlib** format mesh in file `filename`.

`void write(const char * filename)`

Write the current mesh on disk with the **FVlib** format in file `filename`.

`string getName()`

Return the name of the mesh. Void string if no name is defined.

`void setName(const char * name)`

Set the name of the mesh.

`size_t getNbVertex()`

Return the number of vertices of the mesh. Return 0 if no mesh is present.

`size_t getNbCell()`

Return the number of cells of the mesh. Return 0 if no mesh is present.

`FVVertex1D * getVertex(size_t i)`

Return the address of vertex `i`, from 0 to `getNbVertex`-1.

`FVVertex1 * beginBoundaryVertex()` .

Initialize the internal pointer to the first vertex of the mesh which is on the boundary of the domain. Return the address. An edge on the boundary has a `NULL` pointer for the `rigthCell`.

`FVVertex1D * nextBoundaryVertex()`

Return the pointer of the current vertex on the boundary and move to the next one. If we reach the end of the list, return `NULL`.

`FVCell1D * getCell(size_t i)`

Return the address of cell i, from 0 to getNbCell-1.

`FVVertex1D * beginVertex()`

Initialize the internal pointer to the first vertex of the mesh and return the address.

`FVVertex1D * nextVertex()`

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return NULL.

`FVCell1D * beginCell()`

Initialize the internal pointer to the first cell of the mesh and return the address.

`FVCell1D * nextCell()`

Return the pointer of the current cell and move to the next one. If we reach the end of the list, return NULL.

`void msh2FVMesh(Gmsh &mg) *`

Convert a Gmsh into a FVMesh1D format.

Function.

No extern function for this class

Example.

Listing 2.7: Loop over the vertices of a 1D mesh

```
FVMesh1D("my_mesh_1D");
FVVertex1D *ptr_v;
for(m.beginVertex();(ptr_v=m.nextVertex());)
    {
    cout<<"label of vertex "<<ptr_v->label<<endl;
    cout<<"coordinate of the vertex "<<ptr_v->coord<<endl;
    }
```

Listing 2.8: Loop over the cells of a 1D mesh

```
FVMesh1D("my_mesh_1D");
FVCell1D *ptr_c;
for(m.beginCell();(ptr_c=m.nextCell());)
    {
    cout<<"label of cell "<<ptr_c->label<<endl;
    cout<<"length of the cell "<<ptr_c->length<<endl;
    }
```

2.8 FVMesh2D

FVMesh2D Class Reference

Short description 12
Class for the two-dimensional mesh. *Back to index*

Field.

All the fields are `private`.

Method.

`FVMesh2D()`

Default Constructor method.

`FVMesh2D(const char * filename)`

Constructor method which load the **FVlib** format mesh in file `filename`.

`void read(const char * filename)`

Load the **FVlib** format mesh in file `filename`.

`void write(const char * filename)`

Write the current mesh on disk with the **FVlib** format in file `filename`.

`string getName()` .

Return the name of the mesh. Void string if no name is defined.

`void setName(const char * name)`

Set the name of the mesh.

`size_t getNbVertex()` .

Return the number of vertices of the mesh. Return 0 if no mesh is present.

`size_t getNbEdge()` .

Return the number of edges of the mesh. Return 0 if no mesh is present.

`size_t getNbCell()` .

Return the number of cells of the mesh. Return 0 if no mesh is present.

`FVVertex2D * getVertex(size_t i)` .

Return the address of vertex `i`, from 0 to `getNbVertex`-1.

`FVEdge2D * getEdge(size_t i)` .

Return the address of edge `i`, from 0 to `getNbEdge`-1.

`FVCell2D * getCell(size_t i)` .

Return the address of cell `i`, from 0 to `getNbCell`-1.

`FVVertex2D * beginVertex()` .

Initialize the internal pointer to the first vertex of the mesh and return the address.

`FVVertex2D * nextVertex()` .

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

`FVEdge2D * beginEdge()` .

Initialize the internal pointer to the first edge of the mesh and return the address.

`FVEdge2D * nextEdge()` .

Return the pointer of the current edge and move to the next one. If we reach the end of the list, return `NULL`.

`FVEdge2D * beginBoundaryEdge()` .

Initialize the internal pointer to the first edge of the mesh which is on the boundary of the domain. Return the address. An edge on the boundary has a `NULL` pointer for the `rigthCell`.

`FVEdge2D * nextBoundaryEdge()`

Return the pointer of the current edge on the boundary and move to the next one. If we reach the end of the list, return `NULL`.

`FVCell2D * beginCell()` .

Initialize the internal pointer to the first cell of the mesh and return the address.

`FVCell2D * nextCell()` .

Return the pointer of the current cell and move to the next one. If we reach the end of the list, return `NULL`.

`void msh2FVMesh(Gmsh &mg) *`

Convert a `Gmsh` into a `FVMesh2D` format.

Function.

No extern function for this class

Example.

Listing 2.9: Loop over the vertices of a 2D mesh

```
FVMesh2D("my_mesh_2D");
FVVertex2D *ptr_v;
for(m.beginVertex();(ptr_v=m.nextVertex());)
    {
    cout<<"label of vertex "<<ptr_v->label<<endl;
    cout<<"coordinate of the vertex "<<ptr_v->coord<<endl;
    }
```

Listing 2.10: Loop over the edges of a 2D mesh

```
FVMesh2D("my_mesh_2D");
FVEdge2D *ptr_e;
for(m.beginEdge();(ptr_e=m.nextEdge());)
    {
    cout<<"label of edge "<<ptr_e->label<<endl;
    cout<<"coordinate of the centroid "<<ptr_e->centroid<<endl;
    }
```

Listing 2.11: Loop over the cells of a 2D mesh

```
FVMesh2D("my_mesh_2D");
FVCell2D *ptr_c;
for(m.beginCell();(ptr_c=m.nextCell());)
    {
    cout<<"label of cell "<<ptr_c->label<<endl;
    cout<<"surface of the cell "<<ptr_c->area<<endl;
    }
```

2.9 FVMesh3D

FVMesh3D Class Reference

Short description 13
Class for the three-dimensional mesh. *Back to index*

Field.

All the fields are `private`.

Method.

`FVMesh3D()`

Default Constructor method.

`FVMesh3D(const char * filename)`

Constructor method which load the **FVlib** format mesh in file `filename`.

`void read(const char * filename)`

Load the **FVlib** format mesh in file `filename`.

`void write(const char * filename)`

Write the current mesh on disk with the **FVlib** format in file `filename`.

`string getName()`

Return the name of the mesh. Void string if no name is defined.

`void setName(const char * name)`

Set the name of the mesh.

`size_t getNbVertex()`

Return the number of vertices of the mesh. Return 0 if no mesh is present.

`size_t getNbEdge()`

Return the number of edges of the mesh. Return 0 if no mesh is present.

`size_t getNbFace()`

Return the number of faces of the mesh. Return 0 if no mesh is present.

`size_t getNbCell()`

Return the number of cells of the mesh. Return 0 if no mesh is present.

`FVVertex3D * getVertex(size_t i)`

Return the address of vertex `i`, from 0 to `getNbVertex`-1.

`FVEdge3D * getEdge(size_t i)`

Return the address of edge `i`, from 0 to `getNbEdge`-1.

`FVFace3D * getFace(size_t i)`

Return the address of edge `i`, from 0 to `getNbEdge`-1.

`FVCell3D * getCell(size_t i)`

Return the address of cell `i`, from 0 to `getNbCell`-1.

`FVVertex3D * beginVertex()`

Initialize the internal pointer to the first vertex of the mesh and return the address.

`FVVertex3D * nextVertex()`

Return the pointer of the current vertex and move to the next one. If we reach the end of the list, return `NULL`.

`FVEdge3D * beginEdge()`

Initialize the internal pointer to the first edge of the mesh and return the address.

`FVEdge3D * nextEdge()`

Return the pointer of the current edge and move to the next one. If we reach the end of the list, return `NULL`.

`FVFace3D * beginFace()`

Initialize the internal pointer to the first face of the mesh and return the address.

`FVFace3D * nextFace()`

Return the pointer of the current edge and move to the next one. If we reach the end of the list, return `NULL`.

`FVFace3D * beginBoundaryFace()`

Initialize the internal pointer to the first face of the mesh which is on the boundary of the domain. return the address. An edge on the boundary has a `NULL` pointer for the `rigthCell`.

`FVFace3D * nextBoundaryFace()`

Return the pointer of the current face on the boundary and move to the next one. If we reach the end of the list, return `NULL`.

`FVCell3D * beginCell()`

Initialize the internal pointer to the first cell of the mesh and return the address.

`FVCell3D * nextCell()`

Return the pointer of the current cell and move to the next one. If we reach the end of the list, return `NULL`.

`void msh2FVMesh(Gmsh &mg) *`

Convert a `Gmsh` into a `FVMesh3D` format.

Function.

No extern function for this class

Example.

Listing 2.12: Loop over the vertices of a 3D mesh

```
FVMesh3D("my_mesh_3D");
FVVertex3D *ptr_v;
for(m.beginVertex();(ptr_v=m.nextVertex());)
```

```
    {
    cout<<"label of vertex "<<ptr_v->label<<endl;
    cout<<"coordinate of the vertex "<<ptr_v->coord<<endl;
    }
```

Listing 2.13: Loop over the edges of a 3D mesh

```
FVMesh3D("my_mesh_3D");
FVEdge3D *ptr_e;
for(m.beginEdge();(ptr_e=m.nextEdge());)
    {
    cout<<"label of edge "<<ptr_e->label<<endl;
    cout<<"coordinate of the centroid "<<ptr_e->centroid<<endl;
    }
```

Listing 2.14: Loop over the faces of a 3D mesh

```
FVMesh3D("my_mesh_3D");
FVFace3D *ptr_f;
for(m.beginFace();(ptr_f=m.nextFace());)
    {
    cout<<"label of face "<<ptr_f->label<<endl;
    cout<<"coordinate of the centroid "<<ptr_f->centroid<<endl;
    }
```

Listing 2.15: Loop over the cells of a 3D mesh

```
FVMesh3D("my_mesh_3D");
FVCell3D *ptr_c;
for(m.beginCell();(ptr_c=m.nextCell());)
    {
    cout<<"label of cell "<<ptr_c->label<<endl;
    cout<<"volume of the cell "<<ptr_c->volume<<endl;
    }
```

2.10 FVStencil

FVStencil Class Reference

Short description 14
Class to handle stencil. *Back to index*

Field.

No field for this class

Method.

`FVStencil()`

Constructor method.

`~FVStencil()`

Destructor method.

`FVStencil(const FVStencil &)`

Copy constructor method.

`void * beginGeometry()`

Initialize the internal pointer to the first geometrical entity of the stencil and return the address.

`void * nextGeometry()`

Return the pointer of the current geometrical entity and move to the next one. If we reach the end of the list, return `NULL`.

`size_t getNbGeometry()`

Return the number of geometrical entities which compose the stencil.

`void * getGeometry(size_t i)`

Return the pointer of the geometrical entity of index `i`.

`void * getReferenceGeometry()`

Return the pointer of the geometrical entity of reference.

`size_t getType()`

Return the type of the current geometrical entity.

`size_t getType(size_t i)`

Return the type of the geometrical entity of index `i`.

`size_t getReferenceType()`

Return the type of the Reference geometrical entity.

`size_t getIndex()`

Retrun the index of the current geometrical identity.

```
void clean()
```

Clean the stencil. Reset all the variables to zero and pointer to NULL.

```
void show()
```

Print the stencil on the stdout.

```
void addStencil(FVVertex1D *,double w)
```

Add a FVVertex1D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVVertex1D *)
```

Add a FVVertex1D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVVertex1D *)
```

Set the reference geometrical entity as a FVVertex1D.

```
void addStencil(FVVertex2D *,double w))
```

Add a FVVertex2D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVVertex2D *)
```

Add a FVVertex2D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVVertex2D *)
```

Set the reference geometrical entity as a FVVertex2D.

```
void addStencil(FVVertex3D *,double w))
```

Add a FVVertex3D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVVertex3D *)
```

Add a FVVertex3D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVVertex3D *)
```

Set the reference geometrical entity as a FVVertex3D.

```
void addStencil(FVEdge2D *,double w))
```

Add a FVEdge2D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVEdge2D *)
```

Add a FVEdge2D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVEdge2D *)
```

Set the reference geometrical entity as a FVEdge2D.

```
void addStencil(FVEdge3D *,double w))
```

Add a FVEdge3D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVEdge3D *)
```

Add a FVEdge3D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVEdge3D *)
```

Set the reference geometrical entity as a FVEdge3D.

```
void addStencil(FVFace3D *,double w))
```

Add a FVFace3D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVFace3D *)
```

Add a FVFace3D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVFace3D *)
```

Set the reference geometrical entity as a FVFace3D.

```
void addStencil(FVCell1D *,double w))
```

Add a FVCell1D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVCell1D *)
```

Add a FVCell1D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVCell1D *)
```

Set the reference geometrical entity as a FVCell1D.

```
void addStencil(FVCell2D *,double w))
```

Add a FVCell2D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVCell2D *)
```

Add a FVCell2D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVCell2D *)
```

Set the reference geometrical entity as a FVCell2D.

```
void addStencil(FVCell3D *,double w))
```

Add a FVCell3D element to the stencil. Set the weight of the geometrical entity equal to w.

```
void addStencil(FVCell3D *)
```

Add a FVCell3D element to the stencil. The weight is set equal to 1.

```
void setReferenceGeometry(FVCell3D *)
```

Set the reference geometrical entity as a FVCell3D.

```
void addCellStencil(FVMesh1D &m, size_t nbcell)
```

Add to the stencil the closest nbcell FVCell1D of the 1D-mesh m. The distance function is based on the reference geometry centroid (should be set before). If the reference geometrical entity is not set, the origin is use to compute the distance. All the weights are set to 1.

```
void addCellStencil(FVMesh2D &m, size_t nbcell)
```

Add to the stencil the closest nbcell FVCell2D of the 2D-mesh m. The distance function is based on the reference geometry centroid (should be set before). If the reference geometrical entity is not set, the origin is use to compute the distance. All the weights are set to 1.

```
void addCellStencil(FVMesh3D &m, size_t nbcell)
```

Add to the stencil the closest `nbcell` `FVCell3D` of the 3D-mesh `m`. The distance function is based on the reference geometry centroid (should be set before). If the reference geometrical entity is not set, the origin is use to compute the distance. All the weights are set to 1.

```
void addVertexStencil(FVMesh1D &m, size_t nbvertex)
```

Add to the stencil the closest `nbvertex` `FVVeretx1D` of the 1D-mesh `m`. The distance function is based on the reference geometry centroid (should be set before). If the reference geometrical entity is not set, the origin is use to compute the distance. All the weights are set to 1.

```
void addVertexStencil(FVMesh2D &m, size_t nbvertex)
```

Add to the stencil the closest `nbvertex` `FVVeretex2D` of the 2D-mesh `m`. The distance function is based on the reference geometry centroid (should be set before). If the reference geometrical entity is not set, the origin is use to compute the distance. All the weights are set to 1.

```
void addVertexStencil(FVMesh3D &m, size_t nbvertex)
```

Add to the stencil the closest `nbvertex` `FVVertex3D` of the 3D-mesh `m`. The distance function is based on the reference geometry centroid (should be set before). If the reference geometrical entity is not set, the origin is use to compute the distance. All the weights are set to 1.

```
bool inStencil(FVVertex1D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVVertex2D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVVertex3D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVEdge2D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVEdge3D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVFace3D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVCell1D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVCell2D *ptr)
```

Return true if `ptr` is in the stencil else return false.

```
bool inStencil(FVCell3D *ptr)
```

Return true if ptr is in the stencil else return false.

Function.

No function for this class.

Example.

Listing 2.16: handle stencil for a 1D mesh

```
FVMesh1D m;
FVStencil st;
// read a mesh and set the reference geometrical entity
m.read("mesh1D.xml");
st.setReferenceGeometry(m.beginCell());
// gather geometrical entities in the stencil
st.addStencil(m.nextCell());
st.addStencil(m.beginVertex());
st.addStencil(m.nextCell());
st.addStencil(m.nextVertex());
```

2.11 FVRecons1D

FVRecons1D Class Reference

Short description 15

Class to handle the polynomial reconstruction for 1D meshes. *Back to index*

Field.

No field is available in this class.

Method.

`FVRecons1D()`

Constructor method.

`~FVRecons1D()`

Destructor method.

`FVRecons1D(const FVRecons1D &)`

Copy constructor method.

`FVRecons1D(FVStencil *ptr_s, size_t d)`

Constructor method. The method gives the stencil pointer `ptr_s` and the degree `d` for the future polynomial reconstruction.

`FVRecons1D(FVStencil *ptr_s)`

Constructor method. Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`void setStencil(FVStencil *ptr_s, size_t d)`

The method gives the stencil pointer `ptr_s` and the degree `d` for the future polynomial reconstruction.

`void setStencil(FVStencil *ptr_s)`

Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`void setStencil(FVStencil &s, size_t d)`

The method gives the stencil reference `s` and the degree `d` for the future polynomial reconstruction.

`void setStencil(FVStencil &s)`

Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`FVStencil * getStencil()`

Return the stencil associated to the reconstruction. Return `NULL` if no stencil is

allocated.

`void show()`

Print the reconstruction parameters on the stdout.

`void clean()`

Clean the reconstruction. Empty the matrixes and the coefficients. After a `clean`, the user can reuse the variable as a new one.

`void setPolynomialDegree(size_t d)`

Set the polynomial degree of the reconstruction.

`size_t getPolynomialDegree()`

Return the polynomial degree of the reconstruction.

`void setReferencePoint(FVPoint1D<double> P)`

Set the reference point for the polynomial reconstruction.

`void setVectorVertex1D(FVVect<double> & u)`

Give the data associated to the Vextexs of the mesh.

`void setVectorCell1D(FVVect<double> & u)`

Give the data associated to the Cells of the mesh.

`void setReconstructionType(size_t rec_type)`

Set the reconstruction type. Parameter `rec_type` will be set to REC_CONSERVATIVE for a conservative reconstruction and to REC_NON_CONSERVATIVE for a nonconservative reconstruction.

`size_t getReconstructionType()`

Return the reconstruction type. The value is REC_CONSERVATIVE for a conservative reconstruction and REC_NON_CONSERVATIVE for a nonconservative reconstruction.

`void doConservativeMatrix()`

Compute the Matrix and do the QR factorization with respect to the stencil. The weights associated to the geometrical entities of the stencil are used to weight the lines of over-determined linear system. The final polynomial function will respect the mean value with respect to the Reference entity. Stencil must be provided with a reference entity to use the method.

`void doNonConservativeMatrix()`

Compute the Matrix and do the QR factorization with respect to the stencil. The final polynomial function is the interpolation of the values in the Least square meaning. Stencil must be provided to use the method but no reference element is require. Reference Point is the Origin by default bu should be modified with `setReferencePoint`.

`void doMatrix()`

Use methods `doConservativeMatrix` or `doNonConservativeMatrix` in function of the reconstruction type value prescribed by method `setReconstructionType`.

```
void computeConservativeCoef()
```

Compute the coefficient of the conservative polynomial reconstruction. Method `doMatrix` or `doConservativeMatrix` must have been executed once to compute the matrix associated to the vertex. Data must be provided via methods `setVectorVertex1D` or `setVectorCell1D` in function of the geometrical entities used in the stencil.

```
void computeNonConsevativeCoef()
```

Compute the coefficient of the polynomial reconstruction. Method `doMatrix` or `doNonConservativeMatrix` must have been executed once to compute the matrix associated to the vertex. Data must be provided via methods `setVectorVertex1D` or `setVectorCell1D` in function of the geometrical entities used in the stencil.

```
void computeCoef()
```

Use methods `computeConservativeCoef` or `computeNonConservativeCoef` in function of the reconstruction type value prescribed by method `setReconstructionType`.

```
double ConditioningNumber()
```

Compute the conditioning number of the linear system associated to the polynomial reconstruction. Method `doMatrix` or `doConservativeMatrix` must have been executed once to generate the QR decomposition. Then we compute the ratio between the min and the max of the diagonal values. Return 0 if the QR decomposition is not available.

```
double getValue(FVPoint1D<double> P, size_t d)
```

Compute the polynomial at point `P` of degree `d`. `d` must be lower or equal to the degree of the polynomial reconstruction.

```
double getValue(FVPoint1D<double> P)
```

Same thing as above but we assume `d` is the polynomial reconstruction degree.

```
FVPoint1D<double> getDerivative(FVPoint1D<double> P, size_t d)
```

Compute the polynomial derivative at point `P` of degree `d`. `d` must be lower or equal to the degree of the polynomial reconstruction.

```
FVPoint1D<double> getDerivative(FVPoint1D<double> P)
```

Same thing as above but we assume `d` is the polynomial reconstruction degree.

Function.

```
FVPoint1D<size_t> alpha1D(size_t k)
```

Return the power $\alpha(k) = \alpha^1$ associated to the monomial $p(x_1) = (x_1)^{\alpha^1}$ where `k` ranges between 0 and $d-1$, with d the degree of the polynomial reconstruction. This function is the fundamental link between the powers and their storage in a vector.

Example.

2.12 FVRecons2D

FVRecons2D Class Reference

Short description 16

Class to handle the polynomial reconstruction for 2D meshes.

Back to index

Field.

No field is available in this class.

Method.

`FVRecons2D()`

Constructor method.

`~FVRecons2D()`

Destructor method.

`FVRecons2D(const FVRecons2D &)`

Copy constructor method.

`FVRecons2D(FVStencil *ptr_s, size_t d)`

Constructor method. The method gives the stencil pointer `ptr_s` and the degree `d` for the future polynomial reconstruction.

`FVRecons2D(FVStencil *ptr_s)`

Constructor method. Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`void setStencil(FVStencil *ptr_s, size_t d)`

The method gives the stencil pointer `ptr_s` and the degree `d` for the future polynomial reconstruction.

`void setStencil(FVStencil *ptr_s)`

Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`void setStencil(FVStencil &s, size_t d)`

The method gives the stencil reference `s` and the degree `d` for the future polynomial reconstruction.

`void setStencil(FVStencil &s)`

Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`FVStencil * getStencil()`

Return the stencil associated to the reconstruction. Return `NULL` if no stencil is

allocated.

```
void show()
```

Print the reconstruction parameters on the stdout.

```
void clean()
```

Clean the reconstruction. Empty the matrixes and the coefficients. After a `clean`, the user can reuse the variable as a new one.

```
void setPolynomialDegree(size_t d)
```

Set the polynomial degree of the reconstruction.

```
size_t getPolynomialDegree()
```

Return the polynomial degree of the reconstruction.

```
void setReferencePoint(FVPoint2D<double> P)
```

Set the reference point for the polynomial reconstruction.

```
void setVectorVertex2D(FVVect<double> & u)
```

Give the data associated to the Vextexs of the mesh.

```
void setVectorEdge2D(FVVect<double> & u)
```

Give the data associated to the Edges of the mesh.

```
void setVectorCell2D(FVVect<double> & u)
```

Give the data associated to the Cells of the mesh.

```
void setReconstructionType(size_t rec_type)
```

Set the reconstruction type. Parameter `rec_type` will be set to REC_CONSERVATIVE for a conservative reconstruction and to REC_NON_CONSERVATIVE for a nonconservative reconstruction.

```
size_t getReconstructionType()
```

Return the reconstruction type. The value is REC_CONSERVATIVE for a conservative reconstruction and REC_NON_CONSERVATIVE for a nonconservative reconstruction.

```
void doConservativeMatrix()
```

Compute the Matrix and do the QR factorization with respect to the stencil. The weights associated to the geometrical entities of the stencil are used to weight the lines of over-determined linear system. The final polynomial function will respect the mean value with respect to the Reference entity. Stencil must be provided with a reference entity to use the method.

```
void doNonConservativeMatrix()
```

Compute the Matrix and do the QR factorization with respect to the stencil. The final polynomial function is the interpolation of the values in the Least square meaning. Stencil must be provided to use the method but no reference element is require. Reference Point is the Origin by default bu should be modified with `setReferencePoint`.

```
void doMatrix()
```

Use methods doConservativeMatrix or doNonConservativeMatrix in function of the reconstruction type value prescribed by method setReconstructionType.

```
void computeConservativeCoef()
```

Compute the coefficient of the conservative polynomial reconstruction. Method doMatrix or doConservativeMatrix must have been executed once to compute the matrix associated to the vertex. Data must be provided via methods setVectorVertex2D, setVectorEdge2D or setVectorCell12D in function of the geometrical entities used in the stencil.

```
void computeNonConservativeCoef()
```

Compute the coefficient of the polynomial reconstruction. Method doMatrix or doNonConservativeMatrix must have been executed once to compute the matrix associated to the vertex. Data must be provided via methods setVectorVertex2D, setVectorEdge2D or setVectorCell2D in function of the geometrical entities used in the stencil.

```
void computeCoef()
```

Use methods computeConservativeCoef or computeNonConservativeCoef in function of the reconstruction type value prescribed by method setReconstructionType.

```
double ConditioningNumber()
```

Compute the conditioning number of the linear system associated to the polynomial reconstruction. Method doMatrix or doConservativeMatrix must have been executed once to generate the QR decomposition. Then we compute the ratio between the min and the max of the diagonal values. Return 0 if the QR decomposition is not available.

```
double getValue(FVPoint2D<double> P, size_t d)
```

Compute the polynomial at point P of degree d. d must be lower or equal to the degree of the polynomial reconstruction.

```
double getValue(FVPoint2D<double> P)
```

Same thing as above but we assume d is the polynomial reconstruction degree.

```
FVPoint2D<double> getDerivative(FVPoint2D<double> P, size_t d)
```

Compute the polynomial derivatives at point P of degree d. d must be lower or equal to the degree of the polynomial reconstruction.

```
FVPoint2D<double> getDerivative(FVPoint2D<double> P)
```

Same thing as above but we assume d is the polynomial reconstruction degree.

Function.

```
FVPoint2D<size_t> alpha2D(size_t k)
```

Return the power $\alpha(k) = (\alpha^1, \alpha^2)$ associated to the monomial $p(x_1, x_2) = (x_1)^{\alpha^1}(x_2)^{\alpha^2}$ where k ranges between 0 and $d(d+1)/2 - 1$, with d the degree of the polynomial reconstruction. This function is the fundamental link between the powers and their storage in a vector.

Example.

2.13 FVRecons3D

FVRecons3D Class Reference

Short description 17

Class to handle the polynomial reconstruction for 3D meshes. *Back to index*

Field.

No field is available in this class.

Method.

`FVRecons3D()`

Constructor method.

`~FVRecons3D()`

Destructor method.

`FVRecons3D(const FVRecons3D &)`

Copy constructor method.

`FVRecons3D(FVStencil *ptr_s, size_t d)`

Constructor method. The method gives the stencil pointer `ptr_s` and the degree `d` for the future polynomial reconstruction.

`FVRecons3D(FVStencil *ptr_s)`

Constructor method. Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`void setStencil(FVStencil *ptr_s, size_t d)`

The method gives the stencil pointer `ptr_s` and the degree `d` for the future polynomial reconstruction.

`void setStencil(FVStencil *ptr_s)`

Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`void setStencil(FVStencil &s, size_t d)`

The method gives the stencil reference `s` and the degree `d` for the future polynomial reconstruction.

`void setStencil(FVStencil &s)`

Same class as above but `d` is assumed to be zero leading to a first-order reconstruction.

`FVStencil * getStencil()`

Return the stencil associated to the reconstruction. Return `NULL` if no stencil is

allocated.

```
void show()
```

Print the reconstruction parameters on the stdout.

```
void clean()
```

Clean the reconstruction. Empty the matrixes and the coefficients. After a `clean`, the user can reuse the variable as a new one.

```
void setPolynomialDegree(size_t d)
```

Set the polynomial degree of the reconstruction.

```
size_t getPolynomialDegree()
```

Return the polynomial degree of the reconstruction.

```
void setReferencePoint(FVPoint3D<double> P)
```

Set the reference point for the polynomial reconstruction.

```
void setVectorVertex3D(FVVect<double> & u)
```

Give the data associated to the Vextexs of the mesh.

```
void setVectorEdge3D(FVVect<double> & u)
```

Give the data associated to the Edges of the mesh.

```
void setVectorFace3D(FVVect<double> & u)
```

Give the data associated to the Faces of the mesh.

```
void setVectorCell3D(FVVect<double> & u)
```

Give the data associated to the Cells of the mesh.

```
void setReconstructionType(size_t rec_type)
```

Set the reconstruction type. Parameter `rec_type` will be set to REC_CONSERVATIVE for a conservative reconstruction and to REC_NON_CONSERVATIVE for a nonconservative reconstruction.

```
size_t getReconstructionType()
```

Return the reconstruction type. The value is REC_CONSERVATIVE for a conservative reconstruction and REC_NON_CONSERVATIVE for a nonconservative reconstruction.

```
void doConservativeMatrix()
```

Compute the Matrix and do the QR factorization with respect to the stencil. The final polynomial function will respect the mean value with respect to the Reference entity. Stencil must be provided with a reference entity to use the method.

```
void doNonConservativeMatrix()
```

Compute the Matrix and do the QR factorization with respect to the stencil. The weights associated to the geometrical entities of the stencil are used to weight the lines of over-determined linear system. The final polynomial function is the interpolation of the values in the Least square meaning. Stencil must be provided to use the method but no reference element is require. Reference Point is the Origin by default bu should be modified with `setReferencePoint`.

```
void doMatrix()
```

Use methods `doConservativeMatrix` or `doNonConservativeMatrix` in function of the reconstruction type value prescribed by method `setReconstructionType`.

```
void computeConservativeCoef()
```

Compute the coefficient of the conservative polynomial reconstruction. Method `doConservativeMatrix` or `doMatrix` must have been executed once to compute the matrix associated to the vertex. Data must be provided via methods `setVectorVertex3D`, `setVectorEdge3D`, `setVectorFace3D` or `setVectorCell13D` in function of the geometrical entities used in the stencil.

```
void computeNonConsevativeCoef()
```

Compute the coefficient of the polynomial reconstruction. Method `doNonConservativeMatrix` or `doMatrix` must have been executed once to compute the matrix associated to the vertex. Data must be provided via methods `setVectorVertex3D`, `setVectorEdge3D`, `setVectorFace3D`or `setVectorCell3D` in function of the geometrical entities used in the stencil.

```
void computeCoef()
```

Use methods `computeConservativeCoef` or `computeNonConservativeCoef` in function of the reconstruction type value prescribed by method `setReconstructionType`.

```
double ConditioningNumber()
```

Compute the conditioning number of the linear system associated to the polynomial reconstruction. Method `doMatrix` or `doConservativeMatrix` must have been executed once to generate the QR decomposition. Then we compute the ratio between the min and the max of the diagonal values. Return 0 if the QR decomposition is not available.

```
double getValue(FVPoint3D<double> P, size_t d)
```

Compute the polynomial at point `P` of degree `d`. `d` must be lower or equal to the degree of the polynomial reconstruction.

```
double getValue(FVPoint3D<double> P)
```

Same thing as above but we assume `d` is the polynomial reconstruction degree.

```
FVPoint3D<double> getDerivative(FVPoint3D<double> P, size_t d)
```

Compute the polynomial derivatives at point `P` of degree `d`. `d` must be lower or equal to the degree of the polynomial reconstruction.

```
FVPoint3D<double> getDerivative(FVPoint3D<double> P)
```

Same thing as above but we assume `d` is the polynomial reconstruction degree.

Function.

```
FVPoint3D<size_t> alpha3D(size_t k)
```

Return the power $\alpha(k) = (\alpha^1, \alpha^2, \alpha^3)$ associated to the monomial $p(x_1, x_2, x_3) = (x_1)^{\alpha^1}(x_2)^{\alpha^2}(x_3)^{\alpha^3}$ where `k` ranges between 0 and $d(d+1)(d+2)/6 - 1$, with d the degree of the polynomial reconstruction. This function is the fundamental link between the powers and their storage in a vector.

Example.

2.14 FVVertex1D

`FVVertex1D` Class Reference

Short description 18
Class for the one-dimensional vertex. *Back to index*

Field.

`size_t label`

The label of the vertex.

`size_t code`

The code of the vertex.

`FVPoint1D<double> coord`

The x-coordinate of the vertex.

`FVPoint1D<double> normal`

The x-coordinate of normal vector the vertex from left to right. The value is -1 or 1.

`FVCell1D *leftCell,*rightCell`

Pointer to the left and right cell of the edge. `leftCell` is always the inner cell. `rightCell`=`NULL` if the edge is on the boundary.

Method.

`FVCell1D()`

Constructor method.

`~FVCell1D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVVertex1D`. The value is 1 for the vertex.

`double getMeanValue(double (&f))`

Return the value of funtion `f` at the vertex. The function must be declared as `double f(FVPoint1D<double> )`.

`double getMeanValue(double (&f), Parameter &para)`

Return the value of funtion `f` at the vertex. The function must be declared as `double f(FVPoint1D<double>, Parameter &para)`.

`FVCell1D * beginCell()`

Initialize the internal pointer to the first cell in contact with the vertex and return the address.

```
FVCell1D * nextCell()
```

Return the pointer of the current cell and move to the next one. If we reach the end of the list, return `NULL`.

Function.

```
inline bool isEqual(FVVertex1D *c1, FVVertex1D*c2)
```

Return `true` if `c1` and `c2` have the same labels.

Example.

2.15 FVVertex2D

FVVertex2D Class Reference

Short description 19
Class for the two-dimensional vertex. *Back to index*

Field.

`size_t label`

The label of the edge.

`size_t code`

The code of the edge.

`size_t nb_cell`

The number of cells which share the vertex.

`FVPoint2D<double> coord`

The (x, y)-coordinates of the vertex.

`FVCell2D *cell[NB_CELL_PER_VERTEX_2D]`

Pointer table to the cells which share the vertex from 0 to `nb_cell`-1. NB_CELL_PER_VERTEX_2D is the maximum number of cells in contact with the vertex defined in file `FVLib_config.h`.

Method.

`FVVertex2D()`

Constructor method.

`~FVVertex2D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVVertex2D`. The value is 1 for the vertex.

`double getMeanValue(double (&f))`

Return the value of funtion `f` at the vertex. The function must be declared as `double f(FVPoint2D<double> )`.

`double getMeanValue(double (&f), Parameter &para)`

Return the value of funtion `f` at the vertex. The function must be declared as `double f(FVPoint2D<double>, Parameter &para)`.

`FVCell2D * beginCell()`

Initialize the internal pointer to the first cell in contact with the vertex and return the address.

```
FVCell2D * nextCell()
```

Return the pointer of the current cell and move to the next one. If we reach the end of the list, return `NULL`.

Function.

```
inline bool isEqual(FVVertex2D *c1, FVVertex2D*c2)
```

Return `true` if `c1` and `c2` have the same labels.

Example.

2.16 FVVertex3D

FVVertex3D Class Reference

Short description 20
Class for the three-dimensional vertex. *Back to index*

Field.

`size_t label`

The label of the edge.

`size_t code`

The code of the edge.

`size_t nb_cell`

The number of cells which share the vertex.

`FVPoint3D<double> coord`

The (x, y, z)-coordinates of the vertex.

`FVCell3D *cell[NB_CELL_PER_VERTEX_3D]`

Pointer table to the cells which share the vertex from 0 to `nb_cell`-1. NB_CELL_PER_VERTEX_3D is the maximum number of cells in contact with the vertex defined in file `FVLib_config.h`.

Method.

`FVVertex3D()`

Constructor method.

`~FVVertex3D()`

Destructor method.

`double measure()`

Return the measure of the geometrical entity `FVVertex3D`. The value is 1 for the vertex.

`double getMeanValue(double (&f))`

Return the value of funtion `f` at the vertex. The function must be declared as `double f(FVPoint3D<double> )`.

`double getMeanValue(double (&f), Parameter &para)`

Return the value of funtion `f` at the vertex. The function must be declared as `double f(FVPoint3D<double>, Parameter &para)`.

`FVCell3D * beginCell()`

Initialize the internal pointer to the first cell in contact with the vertex and return the address.

```
FVCell3D * nextCell()
```

Return the pointer of the current cell and move to the next one. If we reach the end of the list, return `NULL`.

Function.

```
inline bool isEqual(FVVertex3D *c1, FVVertex3D*c2)
```

Return `true` if `c1` and `c2` have the same labels.

Example.

2.17 GMElement

GMElement Class Reference

Short description 21
Class to handle gmsh element.

Back to index

Field.

`size_t label`

The label of the element.

`size_t code_physical`

The physical code of the element.

`size_t code_elementary`

The elementary code of the element.

`size_t type_element`

The type of the element using the nomenclature of **Gmsh**.

`size_t nb_node`

The number of node associated to the element.

`size_t dim`

The space dimension associted to the element.

`size_t node[GMSH_NB_NODE_PER_ELEMENT]`

Table which contains the label of the element nodes.; GMSH_NB_NODE_PER_ELEMENT is the maximum number of nodes for a `GMElement` defined in file `FVLib_config.h`.

Method.

`GMElement()`

Constructor method.

`~GMElement()`

Destructor method.

Function.

No extern function for this class

Example.

2.18 Gmsh

Gmsh Class Reference

Short description 22
Class to handle gmsh mesh and output for post-processing with gmsh.
Back to index

Field.

All the fields are `private`.

Method.

`Gmsh()`

Default Constructor method.

`Gmsh(const char * filename)`

Constructor method which load the **FVlib** format mesh in file `filename`.

`void readMesh(const char * filename)`

Open a file and load the **Gmsh** format mesh in file `filename`.

`void writeMesh(const char * filename)`

Open a file and write the current mesh on disk with the **Gmsh** format in file `filename`.

`void close()`

Close the current file.

`size_t getNbNode()`

Return the number of nodes of the mesh. Return 0 if no mesh is present.

`size_t getNbElement()`

Return the number of elements of the mesh. Return 0 if no mesh is present.

`size_t getDim()`

Return the dimension of the mesh. Return 0 if no mesh is present.

`void FVMesh2Gmsh(FVMesh1D &ms)` ;

Convert a `FVMesh1D` into a `Gmsh`.

`void FVMesh2Gmsh(FVMesh2D &ms)` ;

Convert a `FVMesh2D` into a `Gmsh`.

`void FVMesh2Gmsh(FVMesh3D &ms)` ;

Convert a `FVMesh3D` into a `Gmsh`.

`FVVertex3D * getNode(const size_t i)` .

Return the pointer of Node `FVVertex3D`([i] from 0 to `getNbNode`-1.

`GMElement * getElement(const size_t i)` .

Return the pointer of element `GMElement`[i] from 0 to `getNbElement`-1.

`void writeVector(FVVect<double> &, const size_t type, const char *name, double time)` .

Write vector in the current **Gmsh** file.

- The file has to be opened and the mesh written with method `writeMesh`. Then, one can write the vector.
- `type` must be `VERTEX` to write a vector of length `getNbNode` (vertex centered value) or `CELL` to write a vector of length `getNbElement` (cell centered value).
- `name` characterizes the field. **Gmsh** deals with the name in the followinf way. In the same file, the fields with the same name are gathered in the same view with their respective time. In that way, it is possible to show a field for different time step. The fields with different name are displayed in different view.
- `time` The time associated to the field.

`void writeVector(FVVect<double> &, FVVect<double> &, const size_t type, const char *name, double time)` .

Write vectors in the current **Gmsh** file. Usualy to show 2D vectorial field. The other parameters are similar to the previous method.

`void writeVector(FVVect<double> &, FVVect<double> &,FVVect<double> &, const size_t type, const char *name, double time)` .

Write vectors in the current **Gmsh** file. Usualy to show 3D vectorial field. The other parameters are similar to the previous method.

Function.

No extern function for this class

Example.

Chapter 3

Vector and Matrix

3.1 FVDenseM

Short description 23
Class for dense matrix. *Back to index*

Field.

`valarray<T_>a`

The matrice. We can directly acces to the matrix coefficients with `a[i]` from 0 to `lenght`-1. Caution. We have a_{ij} =`a[nb_cols*(i-1)+(j-1)]` with $i = 1, ..,$`nb_cols` and $j = 1, ..,$`nb_rows`.

`size_t nb_rows`

The number of rows of the matrix.

`size_t nb_cols`

The number of columns of the matrix.

`size_t lenght`

The length=`nb_rows`×`nb_cols` of the matrix.

Method.

`FVDenseM()`

Default Constructor method.Construct a matrix of length=0.

`FVDenseM(size_t n)`

Constructor method of a `n`×`n` square matrix.

`FVDenseM(size_t n,size_t m)`

Constructor method of a `n`×`m` rectangular matrix.

`FVDenseM(const FVDenseM<T_> &)`

Copy constructor.

`size_t getNbColumns()`

Return the number of colums of the matrix.

`size_t getNbRows()`

Return the number of rows of the matrix.

`size_t getLength()`

Return the number of coefficients of the matrix.

`valarray<T_> * getTab()`

Return the matrix `a` as a tt valarray vector.

`void resize(size_t n)`

Reallocate memory to provide a `n` square matrix.

```
void resize(size_t n, size_t m)
```

Reallocate memory to provide a `n`×`m` rectangular matrix.

```
void setValue(size_t i, size_t j, const T_ & val)
```

Set `a[nb_cols*i+j]=val` where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

```
void addValue(size_t i, size_t j, const T_ & val)
```

Add `a[nb_cols*i+j]+=val` where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

```
T_ getValue(size_t i, size_t j)
```

Return `a[nb_cols*i+j]` where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

```
void setLine(size_t i, FVVect<double> & vec)
```

Set the `i` line of matrix with vector `vec`. Length of the vector must be equal to the number of rows.

```
void setColumn( size_t j, FVVect<double> & vec )
```

Set the `j` column of matrix with vector `vec`. Length of the vector must be equal to the number of lines.

```
void show()
```

Print the matrix elements on the standard output (console).

```
void Gauss(FVVect<double> & b)
```

Perform the resolution of the linear problem using the Gauss with partial pivoting method for a generic `FVVect` vector. Return the solution in `b`.

```
void Gauss(FVPoint2D<double> & b)
```

Perform the resolution of the linear problem using the Gauss with partial pivoting method for a `FVPoint2D` element. Return the solution in `b`.

```
void Gauss(FVPoint3D<double> & b)
```

Perform the resolution of the linear problem using the Gauss with partial pivoting methodfor a `FVPoint3D` element. Return the solution in `b`.

```
void Gauss(FVPoint4D<double> & b)
```

Perform the resolution of the linear problem using the Gauss with partial pivoting methodfor a `FVPoint4D` element. Return the solution in `b`.

```
void LUFactorize()
```

Perform the $A = LU$ factorization for a square matrix A. The U upper matrix substitutes the A upper part including the diagonal while the L lower matrix substitutes the lower part of A. one value diagonal is assumed for L.

```
void ForwardSubstitution(FVVect<double> & b)
```

Perform the $Ly = b$ resolution assuming one value diagonal. At the end, vector `b` contains the solution y

```
void ForwardSubstitution(FVPoint2D<double> & b)
```

Perform the $Ly = b$ resolution assuming one value diagonal. At the end, FVPoint2D b contains the solution y. The dimension of the square matrix must be equal to 2.

```
void ForwardSubstitution(FVPoint3D<double> & b)
```

Perform the $Ly = b$ resolution assuming one value diagonal. At the end, FVPoint3D b contains the solution y. The dimension of the square matrix must be equal to 3.

```
void ForwardSubstitution(FVPoint4D<double> & b)
```

Perform the $Ly = b$ resolution assuming one value diagonal. At the end, FVPoint4D b contains the solution y. The dimension of the square matrix must be equal to 4.

```
void BackwardSubstitution(FVVect<double> & b)
```

Perform the $Ux = b$ resolution. At the end, vector b contains the solution x

```
void BackwardSubstitution(FVPoint2D<double> & b)
```

Perform the $Ux = b$ resolution. At the end, FVPoint2D b contains the solution x The dimension of the square matrix must be equal to 2.

```
void BackwardSubstitution(FVPoint3D<double> & b)
```

Perform the $Ux = b$ resolution. At the end, FVPoint3D b contains the solution x The dimension of the square matrix must be equal to 3.

```
void BackwardSubstitution(FVPoint4D<double> & b)
```

Perform the $Ux = b$ resolution. At the end, FVPoint4D b contains the solution x The dimension of the square matrix must be equal to 4.

Listing 3.1: LU factorization with FVDenseM matrix

```
size_t nb=4;
FVDenseM<double> A(nb);
FVVect<double> b(nb);
// assuming that A and b are initialized
A.LUFactorize();
A.ForwardSubstitution(b); //solve Ly=b
A.BackwardSubstitution(b); //solve Ux=y
```

```
void QRFactorize(FVDenseM<double> & QT)
```

Perform the $A = QR$ factorization for a over-determined matrix A using the Householder method. The R upper matrix substitutes the A upper part including the diagonal the lower part is set to zero. while the QT matrix is an orthogonal which contains **the transposed** matrix Q.

```
void PartialBackwardSubstitution(FVVect<double> &b)
```

Perform the $Ux = b$ resolution with a upper over-determined system. At the end, vector b contains the solution x

Listing 3.2: QR factorization with FVDenseM matrix

```
size_t nr=4,nc=3;
FVDenseM<double> A(nr,nc), QT(nr);
```

```
FVVect<double> b(nr),y(nr);
// assuming that A and b are initialized
A.QRFactorize(QT);
QT.Mult(b,y);  // attention QT is still the transposed matrix
A.PartialBackwardSubstitution(y); //solve Rx=y with R
                                  //upper rectangular matrix over-determined
```

Operator.

`FVDenseM<T_> & operator=(const T_ &)`

`FVDenseM<T_> & operator+=(const FVDenseM<T_> &)`

`FVDenseM<T_> & operator-=(const FVDenseM<T_> &)`

`FVDenseM<T_> & operator/=(const T_ &)`

`FVDenseM<T_> & operator*=(const T_ &)`

`FVDenseM<T_> & operator+=(const T_ &)`

`FVDenseM<T_> & operator-=(const T_ &)`

The following operations are available

Listing 3.3: available operations with FVDenseM matrix

```
FVDenseM<double> A(3,4), B(3,4);
double x;
A=B; // copy constructeur
A=x; // set all the coefficients with x
A+=B; // add B to A
A-=B; // substract B to A
A+=x; // add x to all the coefficients of A
A-=x; // substract x to all the coefficients of A
A*=x; // multiply x to all the coefficients of A
A/=x; // divide all the coefficients of A with x
```

`void Mult(const FVVect<T_> &x,FVVect<T_> &y) const`

Compute $y = Ax$. The length of x must be equal to the number of columns of A. The length of y must be equal to the number of rows of A.

`void TransMult(const FVVect<T_> &,FVVect<T_> &) const`

Compute $y = A^t x$.The length of x must be equal to the number of rows of A. The length of y must be equal to the number of columns of A.

Listing 3.4: available operation with FVDenseM matrix

```
FVDenseM<double> A(3,4);
FVVect<double> x(4),y(3);
A.Mult(x,y); //Compute y=A x
A.TransMult(y,x);  //Compute x=A^t y
```

Function.

No extern function for this class.

Example.

3.2 FVSparseM

Short description 24
Class for spare matrix. *Back to index*

Field.

`size_t nb_rows`

The number of rows of the matrix; the larger row which contains a non-null element.

`size_t nb_cols`

The number of columns of the matrix: the larger column which contains a non-null element.

`size_t lenght`

The number of non null element of the matrix.

Method.

`FVSparseM()`

Default Constructor method.Construct a matrix.

`FVSparseM(const FVSparseM<T_> &)`

Copy constructor.

`size_t getNbColumns()`

Return the number of colums of the matrix.

`size_t getNbRows()`

Return the number of rows of the matrix.

`size_t getLength()`

Return the number of non-null coefficients of the matrix.

`void setValue(size_t i, size_t j, const T_ & val)`

Set `val` in position (i, j) where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

`T_ getValue(size_t i, size_t j)`

Return `val` in position (i, j) where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

`void addValue(size_t i, size_t j, const T_ & val)`

Add `val` in position (i, j) where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

`void multValue(size_t i, size_t j, const T_ & val)`

Multiply `val` in position (i, j) where `i=0,...,nb_rows-1` and `j=0,...,nb_cols-1`.

`void show()`

Print the matrix elements on the standard output (console).

```
void clean()
```

Clean the matrix and empty the memory.

Operator.

```
FVSparseM<T_> & operator/=(const T_ &)
```

```
FVSparseM<T_> & operator*=(const T_ &)
```

```
void Mult(const FVVect<T_> &x,FVVect<T_> &y) const
```

Compute $y = Ax$. The length of x must be equal to the number of columns of A. The length of y must be equal to the number of rows of A.

```
void TransMult(const FVVect<T_> &,FVVect<T_> &) const
```

Compute $y = A^t x$.The length of x must be equal to the number of rows of A. The length of y must be equal to the number of columns of A.

Function.

No extern function for this class.

Example.

3.3 FVVect

Short description 25
Class for vector. *Back to index*

Field.

All the fields are `private`.

Method.

`FVVect<T_>()`

Default Constructor method. Construct a matrix of length=0.

`FVVect<T_>(size_t n)`

Construct a vector of length `n`.

`FVVect<T_>(const FVVect<T_>& )` Copy constructor.

`size_tsize()`

Return the length of vector.

`void resize(size_t n)`

Reallocate memory to provide a `n` length vector.

`<T_> beginElement()`

Set the iterator pointer at the beginning of the vector.

`<T_> nextElement()`

Return the element and move the pointer to the next element. Return NULL if reaching the end of the vector.

`void setValue(size_t i, const T_ & val)`

Set `val` in position i.

`T_ getValue(size_t i)`

Return `val` in position i.

`void show()`

Print the vector elements on the standard output (console).

Operator.

`FVVect<T_> & operator=(const T_ &)`

`FVVect<T_> & operator+=(const FVVect<T_> &)`

`FVVect<T_> & operator-=(const FVVect<T_> &)`

`FVVect<T_> & operator/=(const T_ &)`

`FVVect<T_> & operator*=(const T_ &)`

```
FVVect<T_> & operator+=(const T_ &)
```

```
FVVect<T_> & operator-=(const T_ &)
```

The following operations are available

Listing 3.5: available operations with FVVect

```
FVVect<double> U(3), V(3);
double x;
U=V; // copy constructeur
U=x; // set all the coefficient with x
U+=V; // add V to U
U-=V; // substract V to U
U+=x; // add x to all the coefficient of U
U-=x; // substract x to all the coefficient of U
U*=x; // multiply x to all the coefficient of U
U/=x; // divide all the coefficient of U with x
```

Function.

```
double Dot(FVVect<double> &u, FVVect<double> &v)
```

Compute the dots product of vectors `u`, `v`.

```
double Norm(FVVect<double> &u)
```

Compute the euclidian norm of vector `u`.

```
double NormMax(FVVect<double> &u)
```

Compute the L^∞ norm of vector `u`.

```
double Min(FVVect<double> &u)
```

Return the minimal componant of vector `u`.

```
double Max(FVVect<double> &u)
```

Return the maximal componant of vector `u`.

Example.

Listing 3.6: access to the vectors

```
FVVect<double> U(100), V(100);
double dots=0.;
for(size_t i=0;i<100;i++)
    dots+=U[i]*V[i];
```

3.4 FVSparseV

Short description 26

Class for sparse vector. *Back to index*

Field.

All the fields are `private`.

Method.

`FVSparseV<T_>()`

Default Constructor method. Construct a vector of size=0.

`FVSparseV<T_>(const FVSparseV<T_>&)`

Copy constructor.

`size_t size()`

Return the length of vector: the larger index corresponding to a non-null element

`void setValue(size_t i, const T_ & val)`

Set `val` in position `i`.

`T_ getValue(size_t i)`

Return the value in position `i`. If the position does not corresponds to a stored coefficients, return `static_cast T_ 0`

`void show()`

Print the vector elements on the standard output (console).

`void clean()`

Clean the vector and remove the memory.

Operator.

`FVSparseV<T_> & operator=(const T_ &)`

`FVSparseV<T_> & operator+=(const FVSparseV<T_> &)`

`FVSparseV<T_> & operator-=(const FVSparseV<T_> &)`

`FVSparseV<T_> & operator/=(const T_ &)`

`FVSparseV<T_> & operator*=(const T_ &)`

`FVSparseV<T_> & operator+=(const T_ &)`

`FVSparseV<T_> & operator-=(const T_ &)`

Function.

Example.

3.5 FVKrylov

Short description 27

Class to use Krylov method. *Back to index*

Field.

No field are availlable for this class.

Method.

`FVKrylov()`

Default Constructor method.Construct a matrix of length=0.

`FVKrylov(size_t n)`

Constructor method of a `n`×`n` square matrix.

`FVKrylov(const FVKrylov<T_> &)`

Copy constructor.

Operator.

Function.

No extern function for this class.

Example.

Chapter 4

Tools

4.1 FVGaussPoint1D

Short description 28
Class to determine the Gauss point for integration of a 1D segment. Back to index

Field.

The class contains no field.

Method.

`FVGaussPoint1D()`

Constructor method.

`~FVGaussPoint1D()`

Destructor method.

`size_t getNbPoint(size_t d)`

Return the number of point use for the quadrature formula of order `d`.

`double getWeight(size_t d,size_t p)`

Return the weight for the quadrature formula of order `d` and the `p` points.

`FVPoint2D<double> getPoint(size_t d,size_t p)`

Return the barycentric coordinates for the `p` points for the quadrature formula of order `d`.

Operator.

No operator for this class

Function.

No function are defined for this class.

Example.

Listing 4.1: quadrature formula 1D

```
FVVertex1D v1,v2;
FVGaussPoint1D G1D;
size_t order=3;
// f(double) is a known function we integrate
double integral=0;
FVPoint2D<double> GP;
for(size_t i=0;i<G1D.getNbPoint();i++)
    {
    GP=G1D.getPoint(d,i);
    intergral+=G1D.getWeight(d,i)*f(GP.x*v1.coord+GP.y*v2.coord);
    }
integral*=// length segment;
```

4.2 FVGaussPoint2D

Short description 29
Class to determine the Gauss point for integration of a 2D triangle. *Back to index*

Field.

The class contains no field.

Method.

`FVGaussPoint2D()`

Constructor method.

`~FVGaussPoint2D()`

Destructor method.

`size_t getNbPoint(size_t d)`

Return the number of point use for the quadrature formula of order `d`.

`double getWeight(size_t d,size_t p)`

Return the weight for the quadrature formula of order `d` and the `p` points.

`FVPoint3D<double> getPoint(size_t d,size_t p)`

Return the barycentric coordinates for the `p` points for the quadrature formula of order `d`.

Operator.

No operator for this class

Function.

No function are defined for this class.

Example.

Listing 4.2: quadrature formula 2D

```
FVVertex2D v1,v2,v3;
FVGaussPoint2D G2D;
size_t order=3;
// f(double) is a known function we integrate
double integral=0;
FVPoint3D<double> GP;
for(size_t i=0;i<G2D.getNbPoint();i++)
    {
    GP=G2D.getPoint(d,i);
    intergral+=G2D.getWeight(d,i)*
               f(GP.x*v1.coord+GP.y*v2.coord+GP.z*v3.coord);
    }
integral*=//surface triangle;
```

4.3 FVGaussPoint3D

Short description 30
Class to determine the Gauss point for integration of a 3D tetrahedron.
Back to index

Field.

The class contains no field.

Method.

`FVGaussPoint3D()`

Constructor method.

`~FVGaussPoint3D()`

Destructor method.

`size_t getNbPoint(size_t d)`

Return the number of point use for the quadrature formula of order `d`.

`double getWeight(size_t d,size_t p)`

Return the weight for the quadrature formula of order `d` and the `p` points.

`FVPoint4D<double> getPoint(size_t d,size_t p)`

Return the barycentric coordinates for the `p` points for the quadrature formula of order `d`.

Operator.

No operator for this class

Function.

No function are defined for this class.

Example.

Listing 4.3: quadrature formula 3D

```
FVVertex2D v1,v2,v3,v4;
FVGaussPoint3D G3D;
size_t order=5;
// f(double) is a known function we integrate
double integral=0;
FVPoint4D<double> GP;
for(size_t i=0;i<G3D.getNbPoint();i++)
    {
    GP=G3D.getPoint(d,i);
    intergral+=G3D.getWeight(d,i)*
               f(GP.x*v1.coord+GP.y*v2.coord+GP.z*v3.coord+GP.t*v4.coord);
    }
integral*=//volume tetrahedron;
```

4.4 FVio

Short description 31

Class to read or write vector in files.

Back to index

Field.

All the fields are `private`.

Method.

```
FVio()
```

Constructor method.

```
~FVio()
```

Destructor method.

```
void setTime(double &time)
```

Set the time value.

```
void setName(string &name)
```

Set the name for the current instance.

```
void open(const char *namefile, int mode)
```

Open the file `namefile` to read a file if `mode=FVREAD` or write a file if `mode=FVWRITE`.

```
void close()
```

Close the file.

```
void put(FVVect<double> &u, const double time=0.,const string
&name="noname")
```

Write the vector `u` of cast `double` using the **FVlib** format for the time `time` (default=0.) and name `name` (default="noname"). `time` and `name` are optional. The file must be opened with mode `FVWRITE`.

```
void put(FVVect<double> &u, FVVect<double> &v, const double
time=0.,const string &name="noname")
```

Write vectors `u,v` of cast `double`. Same options as mentioned above.

```
void put(FVVect<double> &u, FVVect<double> &v, FVVect<double>
&w, const double time=0.,const string &name="noname")
```

Write vectors `u,v,w` of cast `double`. Same options as mentioned above.

```
void put(FVVect<FVPoint1D<double> >& u, const double time=0.,
const string &name="noname")
```

Write vectors `u` of cast `FVPoint1D<double>`. Same options as mentioned above.

```
void put(FVVect<FVPoint2D<double> >& u, const double time=0.,
const string &name="noname")
```

Write vectors `u` of cast `FVPoint2D<double>`. Same options as mentioned above.

```
void put(FVVect<FVPoint3D<double> >& u, const double time=0.,
const string &name="noname")
```

Write vectors `u` of cast `FVPoint3D<double>`. Same options as mentioned above.

```
void get(FVVect<double> &u, FVVect<double> &v, FVVect<double>
&w, double &time, string &name);
```

Read the file and get vectors `u,v,w`, the time `time` and the name `name`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, FVVect<double> &v, FVVect<double>
&w, double &time);
```

Read the file and get vectors `u,v,w` and the time `time`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, FVVect<double> &v, FVVect<double>
&w);
```

Read the file and get vectors `u,v,w`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, FVVect<double> &v, double &time,
string &name);
```

Read the file and get vectors `u,v`, the time `time` and the name `name`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, FVVect<double> &v, double &time);
```

Read the file and get vectors `u,v` and the time `time`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, FVVect<double> &v);
```

Read the file and get vectors `u,v`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, double &time, string &name);
```

Read the file and get vectors `u`, the time `time` and the name `name`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u, double &time);
```

Read the file and get vectors `u` and the time `time`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<double> &u);
```

Read the file and get vectors `u`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint1D<double> > &u, double &time, string &name);
```

Read the file and get vectors `u` of cast `FVPoint1D<double>`, the time `time` and the name `name`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint1D<double> > &u, double &time);
```

Read the file and get vectors `u` of cast `FVPoint1D<double>` and the time `time`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint1D<double> > &u);
```

Read the file and get vectors `u` of cast `FVPoint1D<double>`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint2D<double> > &u, double &time, string &name);
```

Read the file and get vectors `u` of cast `FVPoint2D<double>`, the time `time` and the name `name`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint2D<double> > &u, double &time);
```

Read the file and get vectors `u` of cast `FVPoint2D<double>` and the time `time`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint2D<double> > &u);
```

Read the file and get vectors `u` of cast `FVPoint2D<double>`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint3D<double> > &u, double &time, string &name);
```

Read the file and get vectors `u` of cast `FVPoint3D<double>`, the time `time` and the name `name`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint3D<double> > &u, double &time);
```

Read the file and get vectors `u` of cast `FVPoint3D<double>` and the time `time`. The file must be open with the `FVREAD` mode.

```
void get(FVVect<FVPoint3D<double> > &u);
```

Read the file and get vectors `u` of cast `FVPoint3D<double>`. The file must be open with the `FVREAD` mode.

```
size_t getNbVect()
```

Return the current number of componant of the vector after read or write a *xml* file.

Function.

No extern function for this class

Example.

4.5 FVPoint1D

Short description 32
Class to manipulate one-dimensional points such as coordinates. *Back to index*

Field.

`T_ x`

Components of the points. Template `T_` will be a double, a float or a complex number.

Method.

`FVPoint1D()`

Constructor method.

`~FVPoint1D()`

Destructor method.

`FVPoint1D(const FVPoint1D<T_>& )` Copy constructor.

Operator.

`FVPoint1D<T_> & operator=(const T_ &)`

`FVPoint1D<T_> & operator+=(const FVPoint2D<T_> &)`

`FVPoint1D<T_> & operator-=(const FVPoint2D<T_> &)`

`FVPoint1D<T_> & operator/=(const T_ &)`

`FVPoint1D<T_> & operator*=(const T_ &)`

`FVPoint1D<T_> & operator+=(const T_ &)`

`FVPoint1D<T_> & operator-=(const T_ &)`

`void show();`

Display the ome-dimensional point on the current console.

The following operations are available

Listing 4.4: available operations with FVPoint1D

```
FVPoint1D<double> P, Q;
double val;
P=Q; // copy constructeur
P=val; // set all the coefficient with val
P+=Q; // add P to Q
P-=Q; // substract Q to P
P+=val; // add val to all the coefficient of P
P-=val; // substract val to all the coefficient of P
P*=val; // multiply val to all the coefficient of P
```

```
P/=val; // divide all the coefficient of P with val
```

Function.

```
inline double Norm(const FVPoint1D<double> &u)
```

Return Euclidian norm of u.

```
template <class T_> FVPoint1D<T_> operator+ (const FVPoint1D<T_> &a,
const FVPoint1D<T_> &b)
```

Return the FVPoint1D a+b.

```
template <class T_> FVPoint1D<T_> operator- (const FVPoint1D<T_> &a,
const FVPoint2D<T_> &b)
```

Return the FVPoint1D a-b.

```
template <class T_> T_ operator* (const FVPoint1D<T_> &a, const FVPoint1D<T_> &b)
```

Return the template T_ which corresponds to the dot product between a and b.

```
template <class T_> FVPoint1D<T_>operator- (const FVPoint1D<T_> &a)
```

Return the FVPoint1D -a.

```
template <class T_> FVPoint1D<T_>operator+ (const FVPoint1D<T_> &a, const T_ &x)
```

Return the FVPoint1D a+x where we add x to each componants of a.

```
template <class T_> FVPoint1D<T_>operator- (const FVPoint1D<T_> &a, const T_ &x)
```

Return the FVPoint1D a-x where we substract x to each componants of a.

```
template <class T_> FVPoint1D<T_>operator* (const FVPoint1D<T_> &a, const T_ &x)
```

Return the FVPoint1D a*x where we multiply x to each componants of a.

```
template <class T_> FVPoint1D<T_>operator/ (const FVPoint1D<T_> &a, const T_ &x)
```

Return the FVPoint1D a/x where we divide each componants of a with x.

Listing 4.5: available operations with FVPoint1D

```
FVPoint1D<double> P,Q,R;
double val;
P=Q+R;
P=Q-R;
val=P*Q;// the dot product
val=norm(P); the Euclidian norm
P=-Q;
P=Q+val;
P=Q-val;
P=Q*val;
P=Q/val
```

Example.

4.6 FVPoint2D

Short description 33
Class to manipulate two-dimensional points such as coordinates. *Back to index*

Field.

`T_ x,y`

Components of the points. Template `T_` will be a double, a float or a complex number.

Method.

`FVPoint2D()`

Constructor method.

`~FVPoint2D()`

Destructor method.

`FVPoint2D(const FVPoint2D<T_>& )` Copy constructor.

Operator.

`FVPoint2D<T_> & operator=(const T_ &)`

`FVPoint2D<T_> & operator+=(const FVPoint2D<T_> &)`

`FVPoint2D<T_> & operator-=(const FVPoint2D<T_> &)`

`FVPoint2D<T_> & operator/=(const T_ &)`

`FVPoint2D<T_> & operator*=(const T_ &)`

`FVPoint2D<T_> & operator+=(const T_ &)`

`FVPoint2D<T_> & operator-=(const T_ &)`

`void show();`

Display the two-dimensional point on the current console.

The following operations are available

Listing 4.6: available operations with FVPoint2D

```
FVPoint2D<double> P, Q;
double val;
P=Q; // copy constructeur
P=val; // set all the coefficient with val
P+=Q; // add P to Q
P-=Q; // substract Q to P
P+=val; // add val to all the coefficient of P
P-=val; // substract val to all the coefficient of P
P*=val; // multiply val to all the coefficient of P
```

```
P/=val; // divide all the coefficient of P with val
```

Function.

```
inline double Det(const FVPoint2D<double> &u, const FVPoint2D<double> &v)
```

Return the determinant of the 2 × 2 matrix constituted with u and v.

```
inline double Norm(const FVPoint2D<double> &u)
```

Return Euclidian norm of u.

```
template <class T_> FVPoint2D<T_> operator+ (const FVPoint2D<T_> &a,
const FVPoint2D<T_> &b)
```

Return the FVPoint2D a+b.

```
template <class T_> FVPoint2D<T_> operator- (const FVPoint2D<T_> &a,
const FVPoint2D<T_> &b)
```

Return the FVPoint2D a-b.

```
template <class T_> T_ operator* (const FVPoint2D<T_> &a, const FVPoint2D<T_> &b)
```

Return the template T_ which corresponds to the dot product between a and b.

```
template <class T_> FVPoint2D<T_>operator- (const FVPoint2D<T_> &a)
```

Return the FVPoint2D -a.

```
template <class T_> FVPoint2D<T_>operator+ (const FVPoint2D<T_> &a, const T_ &x)
```

Return the FVPoint2D a+x where we add x to each componants of a.

```
template <class T_> FVPoint2D<T_>operator- (const FVPoint2D<T_> &a, const T_ &x)
```

Return the FVPoint2D a-x where we substract x to each componants of a.

```
template <class T_> FVPoint2D<T_>operator* (const FVPoint2D<T_> &a, const T_ &x)
```

Return the FVPoint2D a*x where we multiply x to each componants of a.

```
template <class T_> FVPoint2D<T_>operator/ (const FVPoint2D<T_> &a, const T_ &x)
```

Return the FVPoint2D a/x where we divide each componants of a with x.

Listing 4.7: available operations with FVPoint2D

```
FVPoint2D<double> P,Q,R;
double val;
P=Q+R;
P=Q-R;
val=P*Q;// the dot product
val=det(P,Q);// the determinant
val=norm(P); the Euclidian norm
P=-Q;
P=Q+val;
P=Q-val;
P=Q*val;
P=Q/val
```

Example.

4.7 FVPoint3D

Short description 34
Class to manipulate three-dimensional points such as coordinates. Back to index

Field.

`T_ x,y,z`

Components of the points. Template `T_` will be a double, a float or a complex number.

Method.

`FVPoint3D()`

Constructor method.

`~FVPoint3D()`

Destructor method.

`FVPoint3D(const FVPoint3D<T_>& )` Copy constructor.

Operator.

`FVPoint3D<T_> & operator=(const T_ &)`

`FVPoint3D<T_> & operator+=(const FVPoint3D<T_> &)`

`FVPoint3D<T_> & operator-=(const FVPoint3D<T_> &)`

`FVPoint3D<T_> & operator/=(const T_ &)`

`FVPoint3D<T_> & operator*=(const T_ &)`

`FVPoint3D<T_> & operator+=(const T_ &)`

`FVPoint3D<T_> & operator-=(const T_ &)`

`void show();`

Display the three-dimensional point on the current console.

The following operations are available

Listing 4.8: available operations with FVPoint3D

```
FVPoint3D<double> P, Q;
double val;
P=Q; // copy constructeur
P=val; // set all the coefficient with val
P+=Q; // add P to Q
P-=Q; // substract Q to P
P+=val; // add val to all the coefficient of P
P-=val; // substract val to all the coefficient of P
P*=val; // multiply val to all the coefficient of P
```

```
P/=val; // divide all the coefficient of P with val
```

Function.

```
inline double Det(const FVPoint3D<double> &u, const FVPoint3D<double> &v,
const FVPoint3D<double> &w)
```

Return the determinant of the 3×3 matrix constituted with `u,v` and `w`.

```
inline FVPoint3D<double> CrossProduct(const FVPoint3D<double> &u,
const FVPoint3D<double> &v)
```

Return the cross product `FVPoint3D` constituted with `u` and `v`.

```
inline double Norm(const FVPoint3D<double> &u)
```

Return Euclidian norm of `u`.

```
template <class T_> FVPoint3D<T_> operator+ (const FVPoint3D<T_> &a,
const FVPoint3D<T_> &b)
```

Return the `FVPoint3D` a+b.

```
template <class T_> FVPoint3D<T_> operator- (const FVPoint3D<T_> &a,
const FVPoint3D<T_> &b)
```

Return the `FVPoint3D` a-b.

```
template <class T_> T_ operator* (const FVPoint3D<T_> &a, const FVPoint3D<T_> &b)
```

Return the template `T_` which corresponds to the dot product between `a` and `b`.

```
template <class T_> FVPoint3D<T_>operator- (const FVPoint3D<T_> &a)
```

Return the `FVPoint3D` -a.

```
template <class T_> FVPoint3D<T_>operator+ (const FVPoint3D<T_> &a, const T_ &x)
```

Return the `FVPoint3D` a+x where we add `x` to each componants of `a`.

```
template <class T_> FVPoint3D<T_>operator- (const FVPoint3D<T_> &a, const T_ &x)
```

Return the `FVPoint3D` a-x where we substract `x` to each componants of `a`.

```
template <class T_> FVPoint3D<T_>operator* (const FVPoint3D<T_> &a, const T_ &x)
```

Return the `FVPoint3D` a*x where we multiply `x` to each componants of `a`.

```
template <class T_> FVPoint3D<T_>operator/ (const FVPoint3D<T_> &a, const T_ &x)
```

Return the `FVPoint3D` a/x where we divide each componants of `a` with `x`.

Listing 4.9: available operations with `FVPoint3D`

```
FVPoint3D<double> P,Q,R;
double val;
P=Q+R;
P=Q-R;
val=P*Q;// the dot product
val=det(P,Q);// the determinant
val=norm(P); the Euclidian norm
P=-Q;
P=Q+val;
```

```
P=Q-val;
P=Q*val;
P=Q/val
```

Example.

4.8 FVPoint4D

Short description 35
Class to manipulate four-dimensional points such as coordinates and time.
Back to index

Field.

`T_ x,y,z,t`

Components of the points. Template `T_` will be a double, a float or a complex number.

Method.

`FVPoint4D()`

Constructor method.

`~FVPoint4D()`

Destructor method.

`FVPoint4D(const FVPoint4D<T_>& )` Copy constructor.

Operator.

`FVPoint4D<T_> & operator=(const T_ &)`

`FVPoint4D<T_> & operator+=(const FVPoint4D<T_> &)`

`FVPoint4D<T_> & operator-=(const FVPoint4D<T_> &)`

`FVPoint4D<T_> & operator/=(const T_ &)`

`FVPoint4D<T_> & operator*=(const T_ &)`

`FVPoint4D<T_> & operator+=(const T_ &)`

`FVPoint4D<T_> & operator-=(const T_ &)`

`void show();`

Display the four-dimensional point on the current console.

The following operations are available

Listing 4.10: available operations with FVPoint4D

```
FVPoint4D<double> P, Q;
double val;
P=Q; // copy constructeur
P=val; // set all the coefficient with val
P+=Q; // add P to Q
P-=Q; // substract Q to P
P+=val; // add val to all the coefficient of P
P-=val; // substract val to all the coefficient of P
```

```
P*=val; // multiply val to all the coefficient of P
P/=val; // divide all the coefficient of P with val
```

Function.

```
inline double Norm(const FVPoint4D<double> &u)
```

Return Euclidian norm of u.

```
template <class T_> FVPoint4D<T_> operator+ (const FVPoint4D<T_> &a,
const FVPoint4D<T_> &b)
```

Return the FVPoint4D a+b.

```
template <class T_> FVPoint4D<T_> operator- (const FVPoint4D<T_> &a,
const FVPoint4D<T_> &b)
```

Return the FVPoint4D a-b.

```
template <class T_> T_ operator* (const FVPoint4D<T_> &a, const FVPoint4D<T_> &b)
```

Return the template T_ which corresponds to the dot product between a and b.

```
template <class T_> FVPoint4D<T_>operator- (const FVPoint4D<T_> &a)
```

Return the FVPoint4D -a.

```
template <class T_> FVPoint4D<T_>operator+ (const FVPoint4D<T_> &a, const T_ &x)
```

Return the FVPoint4D a+x where we add x to each componants of a.

```
template <class T_> FVPoint3D<T_>operator- (const FVPoint4D<T_> &a, const T_ &x)
```

Return the FVPoint3D a-x where we substract x to each componants of a.

```
template <class T_> FVPoint4D<T_>operator* (const FVPoint4D<T_> &a, const T_ &x)
```

Return the FVPoint4D a*x where we multiply x to each componants of a.

```
template <class T_> FVPoint4D<T_>operator/ (const FVPoint4D<T_> &a, const T_ &x)
```

Return the FVPoint4D a/x where we divide each componants of a with x.

Listing 4.11: available operations with FVPoint4D

```
FVPoint4D<double> P,Q,R;
double val;
P=Q+R;
P=Q-R;
val=P*Q;// the dot product
val=norm(P); the Euclidian norm
P=-Q;
P=Q+val;
P=Q-val;
P=Q*val;
P=Q/val
```

Example.

4.9 Parameter

Short description 36

Class to read parameter from a `Parameter` *file format.*

Back to index

Field.

All the fields are `private`.

Method.

`Parameter()`

Default Constructor method.

`~Parameter()`

Destructor method.

`Parameter(const char *filename)`

Constructor method. Read and sparse the file `filename`. The file must respect the `Parameter` file format.

`void read(const char *filename)`

Read and sparse the file `filename`. The file must respect the `Parameter` file format.

`void clean()`

Clean the parameter. All the keys are erased.

`double setParameter(const string &key,const string &value)`

Associate the key contained in `key` with the value contained in `value`.

`double setParameter(const char * key,const char * value)`

Associate the key contained in `key` with the value contained in `value`.

`double getDouble(const string &key)`

Return the double value associated to the `key`. If `key` is not present in the file, a error message is displayed.

`int getInteger(const string &key)`

Return the sign integer value associated to the `key`. If `key` is not present in the file, a error message is displayed.

`size_t getUnsigned(const string &key)`

Return the unsigned integer value associated to the `key`. If `key` is not present in the file, a error message is displayed.

`string getString(const string &key)`

Return the string associated to the `key`. If `key` is not present in the file, a error message is displayed.

`double getDouble(const char *key)`

Same function but key is a `const char *` in place of a string.

`int getInteger(const char *key)`

Same function but key is a `const char *` in place of a string.

`size_t getUnsigned(const char *key)`

Same function but key is a `const char *` in place of a string.

`string getstring(const char *key)`

Same function but key is a `const char *` in place of a string.

`void show()`

Show all the pairs (key value).

Function.

No extern function for this class

Example.

Listing 4.12: read and use a Parameter file

```
Parameter param("my_parameters.xml");
double val=param.getDouble("density");
string name=param.getString("FileName");
size_t NbNode=param.getUnsigned("NumberOfNodes");
```

4.10 Table

Short description 37
Class to read and handle tables contained a `Table` *file format.* *Back to index*

Field.

All the fields are `private`.

Method.

`Table()`

Constructor method.

`~Table()`

Destructor method.

`Table(const char *filename)`

Read and sparse the file `filename`. The file must respect the `Table` file format.

`size_t getNbPoints1()`

Return the number of point for the first indice.

`size_t getNbPoints2()`

Return the number of point for the second indice.

`size_t getNbPoints3()`

Return the number of point for the third indice.

`double getMin1()`

Return the minimal value for the first variable.

`double getMin2()`

Return the minimal value for the second variable.

`double getMin13()`

Return the minimal value for the third variable.

`double getMax1()`

Return the maximal value for the first variable.

`double getMax2()`

Return the maximal value for the second variable.

`double getMax3()`

Return the maximal value for the third variable.

`double linearInterpolation(double x)`

Return the interpolate value of a one-dimension `Table`. If `x` is outside of the range, the function return the closest value which belongs to the convex hull.

```
double linearInterpolation(double x,double y)
```

Return the interpolate value of a two-dimension `Table`. If `x,y` are outside of the range, the function return the closest value which belongs to the convex hull.

```
double linearInterpolation(double x,double y,double z)
```

Return the interpolate value of a three-dimension `Table`. If `x,y,z` are outside of the range, the function return the closest value which belongs to the convex hull.

```
double linearExtrapolation(double x)
```

Return the interpolate value of a one-dimension `Table`. If `x` is outside of the range, the function return an extrapolated value.

```
double linearExtrapolation(double x,double y)
```

Return the interpolate value of a two-dimension `Table`. If `x,y` are outside of the range, the function return an extrapolated value.

```
double linearExtrapolation(double x,double y,double z)
```

Return the interpolate value of a three-dimension `Table`. If `x,y,z` are outside of the range, the function return an extrapolated value.

Function.

No extern function for this class

Example.

Chapter 5

Configuration files

5.1 FVLib_config

Include file which contains the #define of **FVlib**.

```
// ------  FVLIB_config.h ------
#ifndef _FVLIB_Config
#define _FVLIB_Config

#define INF_MIN (-1.E+100)
#define SUP_MAX (1.E+100)
#define FVDOUBLE_PRECISION 1.E-17
#define FVKRYLOV_PRECISION 1.E-10
#define FVEPSI 1.E-12
#define FVPRECISION 12
#define FVCHAMP 20
#define FVCHAMPINT 10
#define FVAPAPTATIVE_MAX_DEGREE 5


#define NB_VERTEX_PER_CELL_2D 9
#define NB_EDGE_PER_CELL_2D 9
#define NB_CELL_PER_VERTEX_2D 15  // this value is checked
#define NB_VERTEX_PER_FACE_3D 9
#define NB_EDGE_PER_FACE_3D 9
#define NB_VERTEX_PER_CELL_3D 9
#define NB_FACE_PER_CELL_3D 9
#define NB_CELL_PER_VERTEX_3D 70// this value is checked
#define GMSH_NB_NODE_PER_ELEMENT 9
#define COMSOL_NB_NODE_PER_ELEMENT 9
//#define NB_ENTITY_PER_STENCIL 40

#define MINUS_THREE_DIM  2147483648
#define MINUS_TWO_DIM    1073741824
#define MINUS_ONE_DIM     536870912
#define FVLIB_PI 3.141592653589793238


#include <string>
#include <map>
typedef  std::map<std::string,std::string> StringMap;



enum FVFile{
           FVNULL     =  0,
           FVOK       ,
           FVREAD     ,
           FVWRITE    ,
           FVENDFILE  ,
           FVNOFILE   ,
           FVWRONGDIM ,
           FVERROR    ,
           VERTEX     ,
           CELL       ,
};
enum FVReconstruction{
```

```
            REC_NULL = 0,
            REC_CONSERVATIVE,
            REC_NON_CONSERVATIVE,
};
enum EntityCode{
            NULL_ENTITY=0,
            FVVERTEX1D,
            FVVERTEX2D,
            FVVERTEX3D,
            FVCELL1D,
            FVCELL2D,
            FVCELL3D,
            FVEDGE2D,
            FVEDGE3D,
            FVFACE3D,
};
enum GMSHTypeElement{
            GMSH_EDGE=1,
            GMSH_TRI,
            GMSH_QUAD,
            GMSH_TETRA,
            GMSH_HEXA,
            GMSH_PRISM,
            GMSH_PYRA,
            GMSH_NODE=15,
};
enum BaliseCode{
            BadBaliseFormat=0,
            EndXMLFile,
            NoOpenBalise,
            NoCloseBalise,
            OkOpenBalise,
            OkCloseBalise,
            NoAttribute,
            OkAttribute,
};
#endif // define _FVLIB_Config
```

5.2 FVGlobal

Include file which contains the global variables of **FVlib**.

```
#ifndef _FVGLOBAL
#define _FVGLOBAL

static unsigned int nb_thread=1;  // number of threads using by openMP
extern unsigned int nb_thread;
#endif // define _FVGLOBAL
```

Part II

The finite volume layer