VTK format for FreeFem++ output data

Gregorio Pellegrini

June 26, 2014

Aim

We would like to use **VTK format** to save the solution of a given problem, solved in FreeFem++.
Let's consider the standard Poisson problem

$$\begin{cases} -\Delta u = f & \Omega \\ u = 0 & \partial\Omega \end{cases}$$

If we want to work with the data related with this problem, immediately we understand that we need to save:

- data related to mesh, namely vertexes of the triangle and triangle itself
- the value of the solution on the point of the mesh

General structure of VTK file

We are going to focus our attention on the **simple legacy format**, that is a style of this file format.
Let's see the general structure of such a file:

```
# vtk DataFile Version 3.0          \\ header
output.vtk, Created by Freefem++    \\ title
ASCII                               \\ data type
DATASET Type                        \\ dataset structure
...
CELL_DATA                           \\ dataset attribute
...
POINT_DATA
...
```

General structure of vtk files

- **HEADER**: It describes file version and the identifier of the file:

  ```
  # vtk DataFile Version 2.0
  ```

  where:
  - #, **vtk**, and **DataFile** are fixed *keywords*
  - **Version 2.0** indicates the version of vtk format we are using, the current one is 3.0, but the Version 1.0 and Version 2.0 are compatible with this latest release
- **TITLE**
  - it is a character string terminated by the end-of-line character $\backslash n$.
  - Is a short description of the data, that are going to be stored.
  - at most 256 characters

  DATA TYPE
  - is a single line containing either the keyword *ASCII* or *BINARY*

General structure of vtk files

DATASET STRUCTURE is the section of our file where we can describe the topology(Points and cells) and geometry(points coordinates) of our dataset.
Type is replace by the following keywords, that encode the structure of the dataset

- STRUCTURED_POINTS
- STRUCTURED_GRID
- RECTILINEAR_GRID
- POLYDATA
- UNSTRUCTURED_GRID
- FIELD

each of this type, encode a particular topology of our dataset, thus it depends on the problem to choose the suitable one:

Some detail

Once we have defined the geometry and topology, let's save data linked with this structure.
There are two possibilities:

- with the keyword **POINT_DATA** we are attaching to the points the some values, it fits for saving the solution at the nodes of our mesh
- with the keyword **CELL_DATA** we are linking the data, specified after this keyword, to the cells, that are defined in the dataset section

It's not important if **POINT_DATA** or **CELL_DATA** comes first.

Remark

These keywords allow the vtk-reader to link the data either with CELLS and or with POINTS.
Later we will see actually how to save such data

About DATASET type

The choice of the dataset type is determine by the geometry of the problem. In particular in our situation it's strictly related to the mesh. if we focus our attention on the dataset type

- STRUCTURED_GRID: regular topology and irregular geometry
- RECTILINEAR_GRID: regular topology but geometry only partially regular
- POLYDATA: irregular in both topology and geometry

All these dataset can be use for our purpose, but require a very strict structure of the mesh on which we are working:

- feasible with meshes characterize by "rectangular" elements
- we need a very regular domain Ω
- more flexible than the previous but it has few possibilities to represent geometric objects

We can point out these consideration just looking at the code-line to define such structure.

RECTILINEAR_GRID

RECTILINEAR_GRID, this file support $1D, 2D, 3D$ structured datasets. The dataset format is followed by:

DATASET RECTILINEAR_GRID
DIMESIONS n_x n_y n_z
X_COORDINATES n_x dataType
$x_0 x_{01} \cdots x_{(n_x-1)}$
Y_COORDINATES n_y dataType
$y_0 y_1 \cdots y_{(n_y-1)}$
Z_COORDINATES n_y dataType
$z_0 z_1 \cdots z_{(n_z-1)}$

where

- DIMENSIONS: they has to be grater or equal than 1, and n_x, n_y, n_z are the number of points in x-y-z direction
- {X,Y,Z}-COORDINATES: are $x - y - z$ coordinates of the points

UNSTRUCTURED_GRID

For our purposes we need a different dataset type:

UNSTRUCTURED_GRID

It is an arbitrary combination of cell type, thus is more flexible. Let's see how define such a dataset type

```
DATASET UNSTRUCTURED_GRID
POINTS n dataType
...
CELL n size
...
CELL_TYPE n
```

Here we can see the flexibility of this approach:

- define the points `POINTS`
- how to connect them `CELL`
- and how to put together these connections `CELL_TYPES`

UNSTRUCTURED_GRID

Let's see in detail what these keywords require:

```
POINTS n dataType
```
p_{0x} p_{0y} p_{0z}

...

$p_{(n-1)x}$ $p_{(n-1)y}$ $p_{(n-1)z}$

where

- `n` is the number of points
- `dataType` is the type of data: *float*, *int*, etc.
- $p_i = (p_{ix}\ p_{iy}\ p_{iz})$ are the coordinates of the i-th points.

UNSTRUCTURED_GRID

Let's see in detail what these keywords require:

`CELL n size`
$numPoints_0 \;\; i_0 \;\; j_0 \;\; k_0$
$numPoints_1 \;\; i_1 \;\; j_1 \;\; k_1$
...
$numPoints_{n-1} \;\; i_{n-1} \;\; j_{n-1} \;\; k_{n-1}$

where

- `n` is the number of cells
- `size` is the cell list size, i.e the total number of integer values required to represent the list
- $numPoints_m$ is the number of point required to define this polygon
- $i_m \; j_m \; k_m$ are the pointer identifies the points of the geometry section.

UNSTRUCTURED_GRID

Let's see the parameter for the keyword `CELL_TYPES`

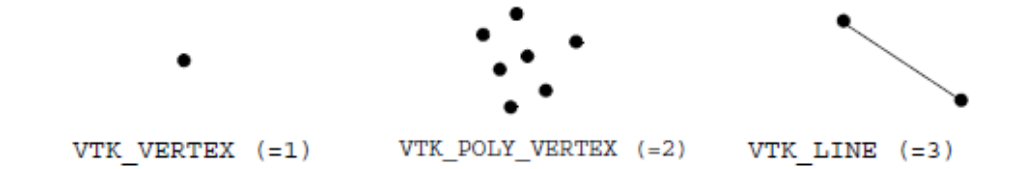
```
CELL_TYPES n
type_0
type_1
...
type_{n-1}
```

in this contest:

- n is the number of cells
- $\texttt{type}_m$ is a natural number chosen from a preset table

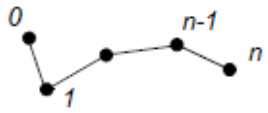
Remark

Here is encode the flexibility of this approach. You can customize every feature of the data: you specify the points, how to collect them, and what is the geometrical object you're building up

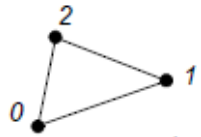

VTK_VERTEX (=1)

VTK_POLY_VERTEX (=2)

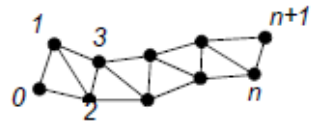
VTK_LINE (=3)



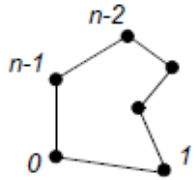
VTK_POLY_LINE (=4)



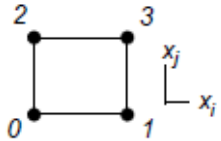
VTK_TRIANGLE(=5)



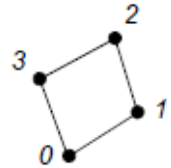
VTK_TRIANGLE_STRIP (=6)



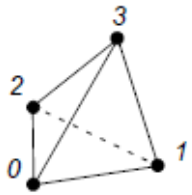
VTK_POLYGON (=7)



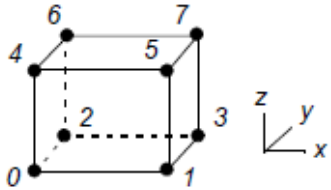
VTK_PIXEL (=8)



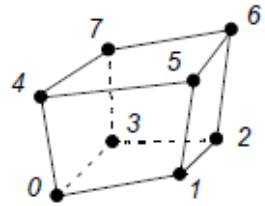
VTK_QUAD (=9)



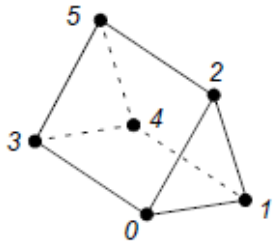
VTK_TETRA (=10)



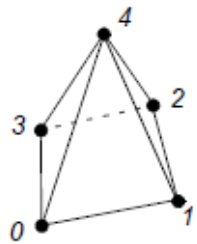
VTK_VOXEL (=11)



VTK_HEXAHEDRON (=12)



VTK_WEDGE (=13)



VTK_PYRAMID (=14)

Dataset attribute

Once we have defined the geometry and topology of our data, let's attach to them the values of quantity of interest.
In our case they are the values of the solution, and we require to save scalar fields, vector fields. These kind of data appears in vtk-format:

- SCALAR
- VECTOR

but actually what we are going to use is a more general dataset attribute:

FIELD DATA

it doesn't introduce any difference with respect to vector or scalar dataset attribute, but it allows to write in the same environment, data that are either scalar or vectors.

Field data

It's essentially an array of data arrays. Defining a field data means giving a name to the field and specifying the number of arrays it contains. Let's see the code line, to define it

```
FIELD dataName numArray
arrayName0 numComponents numTuple dataType
```

$$
\begin{array}{llll}
f_{0,0} & f_{0,1} & \cdots & f_{0,numComponents-1} \\
f_{0,0} & f_{0,1} & \cdots & f_{1,numComponents-1} \\
\vdots & & & \\
f_{(numTuple-1),0} & f_{(numTuple-1),1} & \cdots & f_{(numTuple-1),(numComponents-1)}
\end{array}
$$

$\vdots$

```
arrayName(numArray-1) numComponents numTuple dataType
```

$$
\begin{array}{llll}
f_{0,0} & f_{0,1} & \cdots & f_{0,numComponents-1} \\
f_{0,0} & f_{0,1} & \cdots & f_{1,numComponents-1} \\
\vdots & & & \\
f_{(numTuple-1),0} & f_{(numTuple-1),1} & \cdots & f_{(numTuple-1),(numComponents-1)}
\end{array}
$$

Implementation in Freefem++

We need to save all required data on file, and the Output stream class to operate on files is *ofstream*:

```
{
ofstream file("fileName.vtk");
file << "Hello! I'm saving this string on file" << endl;
}
```

In order to visualize the solution in a vtk-reader we need to save:

- geometry are the points on which the solution is defined
- topology of the problem is completely encode in the mesh **Th**
- the solution of the problem, namely its values on the nodes.

Header

Let's consider our standard Poisson problem, already solved in Freefem++:

$$\begin{cases} \Delta u = 0 & \Omega \\ u = f & \partial\Omega \end{cases}$$

Due to our implementation is not require to know a priori the underlying finite element space **Xh**.
Before starting let's save the values of the nodes:

$$Xh[int]\ xh(2); \qquad xh[0] = x; \qquad xh[1] = y;$$

Let's set the header and the basic information about our vtk file.

```
file ≪ "# vtk DataFile Version 2.0 "≪ endl;
file ≪ "vtk format, created via Freefem++"≪"\n ";
file ≪ "ASCII" ≪ endl;
file ≪ "DATASET UNSTRUCTURED_GRID " ≪ endl;
```

"Assembly of the file: POINTS"

I use the `UNSTRUCTURED_GRID` type of dataset, since is the most general. After the keyword `POINTS` I save all points on which the solution is known.

```
file ≪"POINTS " ≪ Xh.ndof ≪ " float" ≪ endl ;
for (int i = 0 ; i < Xh.ndof ; i++)
{
 file≪xh[0][][i]≪" "≪xh[1][][i]≪" "≪ " 0 " ≪ endl;
}
file ≪ endl;
```

If we are dealing with $3D$ problem we have to save also the z-component of the nodes

$$Xh[int]\ xh(3); \qquad xh[0] = x; \qquad xh[1] = y; \qquad xh[2] = z;$$

instead setting the third component to " 0 " we write `xh[2][][i]`.

"Assembly of the file: CELLS & CELL_TYPE"

```
file ≪"CELLS "≪ Th.nt ≪" "≪ 4*Th.nt≪ endl;
for ( int i = 0 ; i < Th.nt ; i++ ) {
      file ≪ " 3 " ≪ " "; \\ since we have 3 vertices
      for ( int j = 0 ; j < 3 ; j++) {
         file ≪ Xh(i,j) ≪ " ";
      }
      file ≪ endl;
   }
file ≪ "CELL_TYPES " ≪ Th.nt ≪ endl;
for ( int i = 0 ; i < Th.nt ; i++) {
         file ≪ "5 ";
   }
      file ≪ endl;
```

'Assembly of the file"

Let's suppose that the problem that we are going to solve is the usual 2-dimensional possion problem $-\Delta u = f$. Thus we have only one function to keep $u(x, y)$. We save this value using a *FIELD* data attribure

```
file ≪ "POINT_DATA" ≪ Xh.ndof ≪ endl;
file << "FIELD"; << " fieldata "; << " 1 "; << endl;
file ≪ "solution " ≪ " 1 " ≪ Xh.ndof ≪" float" ≪ endl;
for ( int i =0 ; i < Xh.ndof ; i++)
{
   real Thx,Thy; Thx =xh[0][][i]; Thy = xh[1][][i];
   file ≪ u(Thx,Thy) ≪ endl;
}
```

if we are in $3D$ setting, we add the z-component:

```
real Thz; Thy = xh[2][][i];
     u(Thx,Thy,Thz);
```