Rapport Développement Logiciel

Badr YOUBI IDRISSI

1er mai 2018

1 Introduction et fonctionnement général

Dans ce projet nous codons un programme pour visualiser les isolignes d'un signal émit de plusieurs tétraèdres.

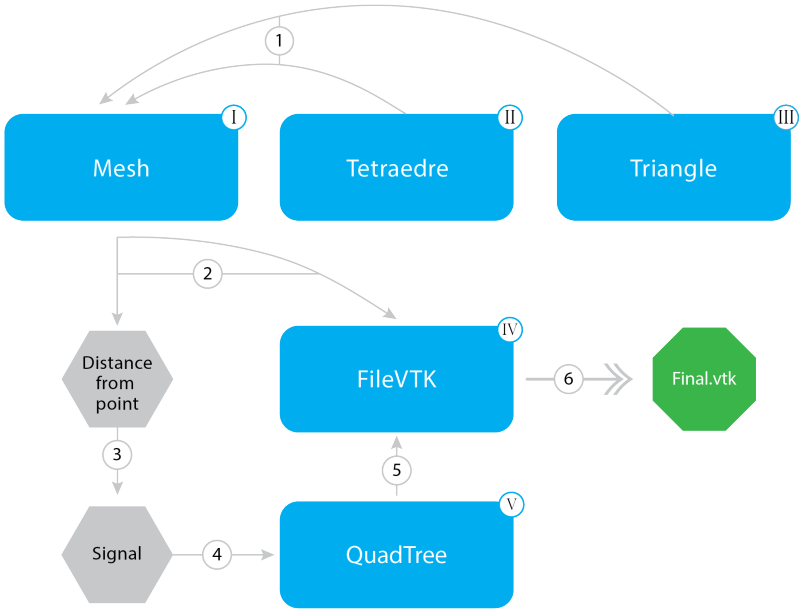

FIGURE 1 – Schéma du fonctionnement général

On commence par créer l'objet 'Mesh' (I) qui correspond à l'ensemble des pods émétant le signal. L'objet mesh a besoin des points correspondant à chaque pod qui est un 'Tétraèdre' (II) qui lui même est un ensemble de 'Triangles' (III) (Étape 1). Cet objet Mesh a la méthode distance from point qui renvoie la liste des distances entre un point et les pods (Étape 2). Ces distances servent ensuite à calculer le signal en tout point de l'espace (Étape 3). on crée ensuite un plan à la hauteur 0.01 dont on va rafiner le maillage avec l'objet 'QuadTree' (V) (Étape 4). Ensuite l'ensemble des points et les valeurs du signal en ces points est passé à l'objet 'FileVTK' (IV) (Étape 5) qui va écrire le fichier "Final.vtk" (Étape 6).

2 Description du code

2.1 Objet III : Triangle

```
class tri:
        def __init__(self, point1, point2, point3):
                ...
        def distance_to_a_point(self, P):
                ...
```

L'objet triangle sert à stocker les points du triangle et à calculer la distance entre un point et à calculer la distance entre un point quelconque et le triangle.

```
def __init__(self, point1, point2, point3):
        self.points = [np.array(point1), np.array(point2), np.array(point3)]
        p1p2 = self.points[1]-self.points[0]
        p1p3 = self.points[2]-self.points[0]
        self.normal = np.cross(p1p2, p1p3)
        self.normal = self.normal/np.linalg.norm(self.normal)
```

On enregistre les points A, B et C et on calcule la normale $\overrightarrow{n'} = \overrightarrow{AB} \times \overrightarrow{AC}$ puis $\vec{n} = \frac{\overrightarrow{n'}}{\|\overrightarrow{n'}\|}$

Pour la fonction de calcul de distance la fonction projète le point sur le plab du triangle puis selon les valeurs des coordonnées barycentriques on retourne la plus petite distance à un coin ou à un bord. Au vu du grand nombre de cas, j'ai opté pour reprendre un code déjà fait qui applique ce principe (voir code) en l'adaptant en rajoutant la condition

```
normal_side  = np.dot(self.normal, P) > np.dot(self.normal,self.points[0])
```

car un point est du bon coté du plan du triangle si $\langle \vec{p}, \vec{n} \rangle > \langle \vec{A}, \vec{n} \rangle$

2.2 Objet II : Tetra

```
class tetra:
        def __init__(self, point1, point2, point3, point4):
                ...
        def distance_to_a_point(self, p):
                ...
```

L'objet tetra est un ensemble de triangles correctement orientés (normales vers l'extérieur) qui peut calculer la distance entre un point et le tetra.

```
def __init__(self, point1, point2, point3, point4):
        self.points = [point1, point2,point3,point4]
        self.faces = [tri(point1,point3,point2), tri(point1,point4,point3),
                                tri(point2,point3,point4), tri(point1,point2,point4)]
```

L'ordre des points ici est important pour bien orienter les normales.

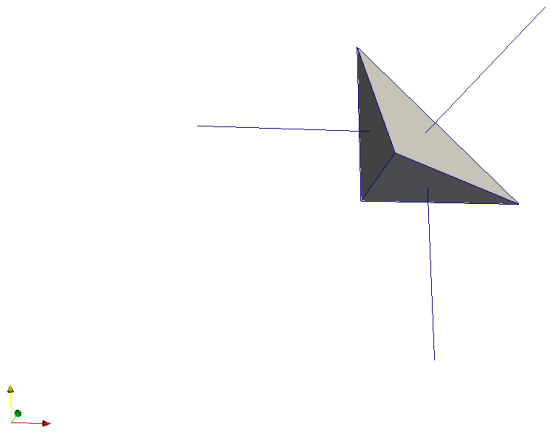

FIGURE 2 – Tétraèdre avec normales

```
def distance_to_a_point(self, p):
        d_min = np.inf
        outside = False

        for f in self.faces:
                [normal_side, d_minTri] = f.distance_to_a_point(p)
                outside = outside or normal_side
                if d_minTri < d_min:
                        d_min = d_minTri

        return [outside, d_min]
```

la distance à un Tetraèdre est le minimum de distance aux autres triangles. sachant que si le point est du mauvais côté la distance est infini.

2.3 Objet I : mesh

```
class mesh:
        def __init__ (self, DefenseFile):
                ...
        def getNextValue(self)::
                ...
        def toVector(self,string):
                ...
        def radialVect(self,angle_in_degrees):
                ...
        def Create_Mesh_And_Connectivity_List(self, List_Of_Point, Connectivity):
                ...
        def Distance_Between_a_Point_and_the_Modules(self, p):
                ...
```

L'objet mesh génère des Tétraèdres à partir du fichier 'defense_zone.txt' qui en donne les centres des bases équilatérales et les hauteur. Les fonctions getNextValue, toVector et radialVect sont des fonctions auxiliaires.

```
def __init__ (self, DefenseFile):
        self.List_Of_Modules = []
        self.file = open(DefenseFile)
        self.number_modules = int(self.getNextValue())

        for i in range(self.number_modules):
                center = self.toVector(self.getNextValue())
                base_length = float(self.getNextValue())
                height = float(self.getNextValue())
                rotation = float(self.getNextValue())
                i_z = np.array([0.0,0.0,1.0])

                distanceFromCenter = (np.sqrt(3)/3)*base_length
                point1 = center + self.radialVect(rotation)*distanceFromCenter
                point2 = center + self.radialVect(rotation + 120)*distanceFromCenter
                point3 = center + self.radialVect(rotation + 240)*distanceFromCenter
                point4 = center + height*i_z

                self.List_Of_Modules.append(tetra(point1, point2, point3, point4))

def getNextValue(self):
        return self.file.readline().strip().split(" : ")[1]
```

```
def radialVect(self,angle_in_degrees):
        i_x = np.array([1.0,0.0,0.0])
        i_y = np.array([0.0,1.0,0.0])
        return np.cos(angle_in_degrees*np.pi/180)*i_x + np.sin(angle_in_degrees*np.pi/180)*i_y
```

Pour trouver les coordonnées du tétraèdres à partir des informations données on commence par trouver la distance entre le centre et chacun des points de la base qui est égale à $\frac{\sqrt{3}}{3}d$ avec d la longueur de la base. Le triangle est donc 3 points équidistants sur le cercle de rayon $\frac{\sqrt{3}}{3}d$. On calcule le vecteur radial avec radialVect. getNextValue lit la ligne suivante et enlève le saut de ligne et donne la valeur après les deux points.

```
def Distance_Between_a_Point_and_the_Modules(self, p):

        outside = True
        d_min = []

        for t in self.List_Of_Modules:
                [outsidet, d_mint] = t.distance_to_a_point(p)
                outside = outside and outsidet
                d_min.append(d_mint)

        return [outside, np.array(d_min)]
```

Cette fonction calcule la liste des distances et si oui ou non le points et dans un des pods.

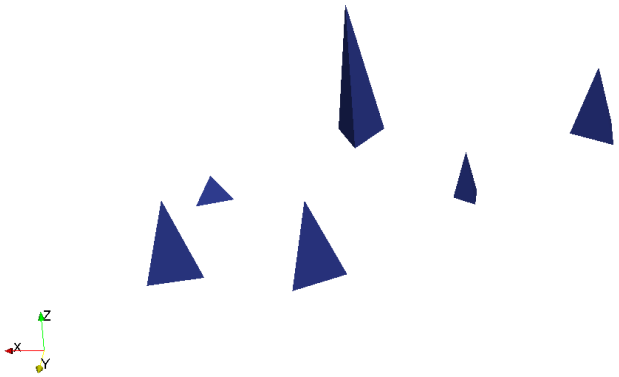

FIGURE 3 – Objet mesh visualisé

2.4 Objet V : QuadTree

L'objet QuadTree est un objet qui raffine un maillage 2D selon les zones d'interêt.

```
class tree_amr:
        def __init__(self, point_0, point_1, point_2, point_3, depth, depth_max, dico):
                ...
        def Get_values(self, List_Of_Values):
                ...
```

Le gros du fonctionnement est dans l'initialisation :

```
self.branch = False

#    store the information locally in the memory for the leaf
self.point_0 = point_0
self.point_1 = point_1
self.point_2 = point_2
self.point_3 = point_3
self.depth_max = depth_max
self.depth = depth
self.dico = dico

r0 = self.dico['eval_function'](self.point_0,self.dico)
r1 = self.dico['eval_function'](self.point_1,self.dico)
r2 = self.dico['eval_function'](self.point_2,self.dico)
r3 = self.dico['eval_function'](self.point_3,self.dico)

self.val0 = r0
self.val1 = r1
self.val2 = r2
self.val3 = r3

if (depth < depth_max):

#   if the depth of the leaf is lower that the maximum depth
#   then we check if we need to transform the leaf into a branch
#   We must transform the leaf into a branch

  if self.dico['refine_or_not'](r0,r1,r2,r3, dico) or depth <= self.dico['level_min']:

  self.branch = True

  point_4 = (point_0+point_2) / 2
  point_5 = (point_0+point_1) / 2
  point_6 = (point_1+point_2) / 2
  point_7 = (point_2+point_3) / 2
  point_8 = (point_0+point_3) / 2

  self.child_up_left = tree_amr(point_8, point_4, point_7, point_3, depth+1, depth_max, dico)
  self.child_up_right = tree_amr(point_4, point_6, point_2, point_7, depth+1, depth_max, dico)
  self.child_down_left = tree_amr(point_0, point_5, point_4, point_8, depth+1, depth_max, dico)
  self.child_down_right = tree_amr(point_5, point_1, point_6, point_4, depth+1, depth_max, dico)
```

On commence par calculer la fonction d'interêt en les coins puis on regarde si ces valeurs nous interessent et si oui on crée récursivement des branches en subdivisant le plan en 4. Tout ceci en vérifiant qu'on ne dépasse pas la profondeur maximale et qu'on dépasse au moins la profondeur minimale.

La fonction qui nous interesse ici est le signal en un point de l'espace.

```
  def signal(p, dico):
      outside, d_min = dico['defZone'].Distance_Between_a_Point_and_the_Modules(p)
          if outside:
                  return sum(1/d_min)
          else:
                  return 2*max(isoligne)
```

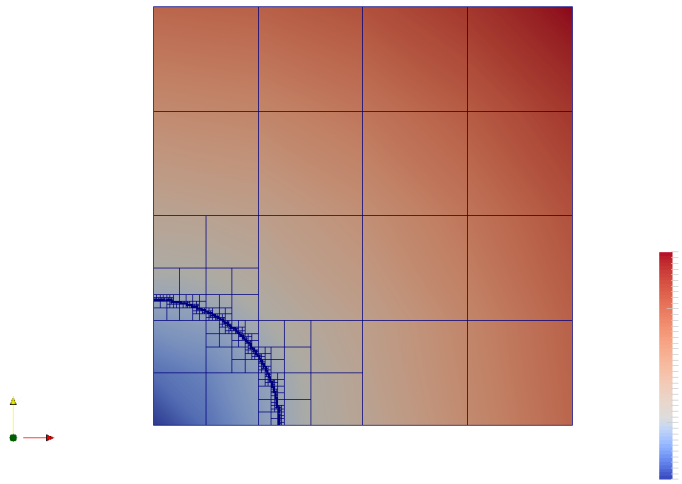

FIGURE 4 – Test du quadtree

La fonction qui définit si oui ou non le carré actuel est un point d'interêt est

```
def test_function(r0, r1, r2, r3, dico):
    cond = False
    for iso in dico['r_target']:
        cond = cond or ((r0 <  iso) | (r1 <  iso) | (r2 <  iso) | (r3 <  iso)) \
                    & ((r0 >= iso) | (r1 >= iso) | (r2 >= iso) | (r3 >= iso))
    return cond
```

cette fonction regarde si la valeur d'une des isolignes est comprise entre les valeurs des coins du carré.

2.5 Objet IV : FileVTK

Cette objet gère l'enregistrement des données sous la forme d'un fichier au format vtk qui permet de visualiser sous ParaView.

```
class FileVTK:

    def __init__(self, FileName, Comment):
                ...
    def SavePoints(self, PointList):
        ...
    def SaveConnectivity(self, Connectivity):
                ...
        def CreatePointScalarSection(self,Points):
                ...
    def SavePointScalar(self, Label, Values):
                ...
    def close(self):
        ...
```

On commence par enregistrer les variables necessaire dans l'objet et on initialise le fichier VTK.

```
def __init__(self, FileName, Comment):
        self.FileName = FileName+".vtk"
        self.file = open(self.FileName, "w")
        self.writel("# vtk DataFile Version 3.1")
        self.writel(Comment)
```

```
        self.writel("ASCII")
        print("File: " + self.FileName + " Opened and initiated")
```

Ensuite on sauvegarde dans le fichier la partie correspondant aux points

```
def SavePoints(self, PointList):
        self.points=PointList
        self.writel("DATASET UNSTRUCTURED_GRID")
        self.writel("POINTS " + str(len(self.points)) + " FLOAT")
        for point in self.points:
                self.writel("  "+str(point[0])+" "+str(point[1])+" "+str(point[2]))
        print("Points saved in "+self.FileName)
```

Puis on enregistre la partie correspondant aux connectivités.

```
def SaveConnectivity(self, Connectivity):
        size = 0
        for c in Connectivity:
                size += len(c)
        self.writel("CELLS "+str(len(Connectivity))+" "+str(size))
        for c in Connectivity:
                self.file.write("  ")
                for i in range(len(c)):
                        if i == 0:
                                self.file.write(str(len(c)-1))
                                self.file.write(" ")
                        else:
                                self.file.write(str(c[i]))
                                self.file.write(" ")
                self.file.write("\n")
        self.writel("CELL_TYPES "+str(len(Connectivity)))
        for c in Connectivity:
                n = len(c)-1
                if c[0] == "2D":
                        if n == 3:
                                celltype = 5
                        if n == 4:
                                celltype = 9
                        if n == 2:
                                celltype = 3
                else:
                        if n == 4:
                                celltype = 10
                        if n == 8:
                                celltype = 12
                self.writel("  "+str(celltype))
        print("Connectivity saved")
```

Ces deux fonctions enregistrent la partie correspondant aux valeurs associées au points.

```
def CreatePointScalarSection(self,Points):
        self.writel("POINT_DATA "+str(len(Points)))

def SavePointScalar(self, Label, Values):
        self.writel("SCALARS "+Label+" float")
        self.writel("LOOKUP_TABLE default")
        for v in Values:
                self.writel("  "+str(v))
```

Cette fonction ferme le fichier.

```
def close(self):
        self.file.close()
        print("File Closed")
```

Tout les autres objets ont une fonction Create_Mesh_And_Connectivity_List qui rajoute à la liste List_Of_Point la liste des points que l'objet génére et à la liste Connectivity comment les relier dans Paraview.

3 Résutat

Le programme marche bien et donne ceci pour les isolignes 1.1 1.2 et 1.5

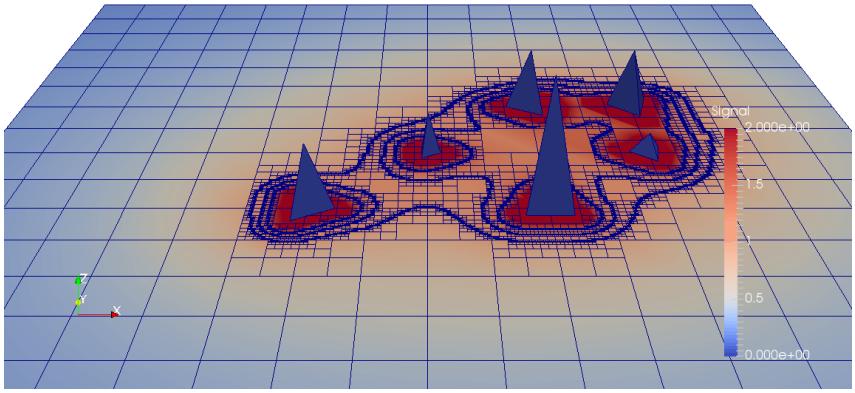

FIGURE 5 – Résultat

4 Conclusion