BOURQUIN Armance
Avril 2018

Rapport du projet de développement logiciel

La classe FileVTK :

La classe FileVTK va nous permettre d'écrire directement depuis Python des fichiers de format vtk que Paraview pourra ensuite lire : c'est ainsi que nous pourrons visualiser nos figures. Pour définir cette classe, nous nous sommes servis de exemple_cas1.vtk que nous avons généralisé.

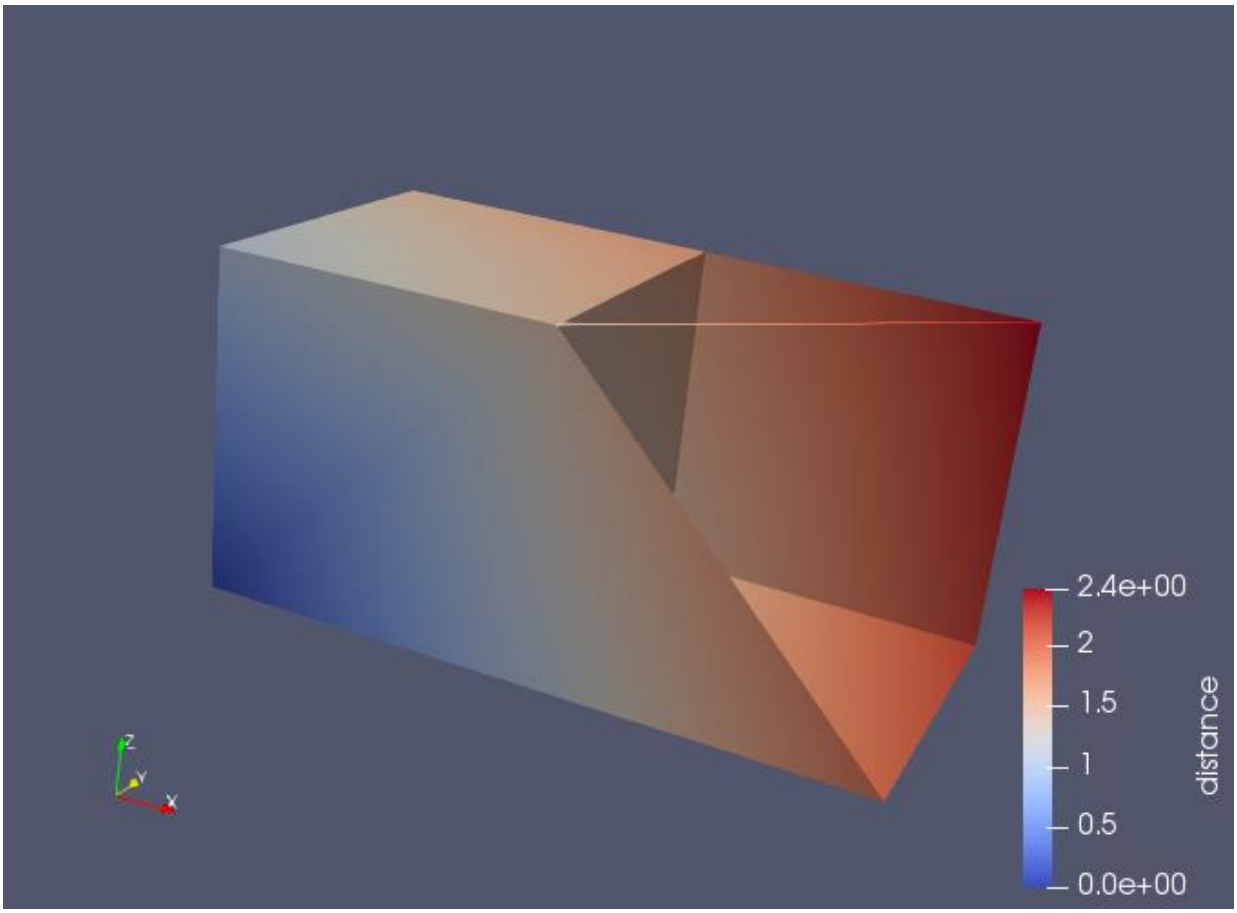
1. **La fonction __init__ :**
   Input : FileName, Comment
   La fonction crée un fichier nommé FileName.vtk et écrit le Comment ainsi que les informations toujours présentes au début des fichiers vtk que nous manipulons. On définit ainsi un attribut ficher, grâce à open et l'on écrit dedans grâce à write.

2. **La fonction SavePoints :**
   Input : FilePoints (liste de points représentés par leurs coordonnées)
   On écrit dans notre fichier, après avoir renseigné leur nombre, les coordonnées de points fournies par FilePoints en passant simplement à la ligne à chaque nouveau point. Cependant, on attribue ainsi implicitement un numéro à chaque point (1 pour le premier déclaré, 2 pour le second et ainsi de suite), numéros que les fonctions suivantes vont utiliser.

3. **La fonction SaveConnectivity :**
   Input : Connectivity (liste de cellules désignées par leur dimension et les points les composant repérés par leur position dans FilePoints)
   On écrit dans notre fichier le nombre de cellules puis pour chaque figure, les numéros renvoyant aux points de cette figure. Ensuite, on liste dans le fichier les numéros associés à chaque cellule renseignée selon la documentation fournie dans le sujet (on ne déclare pas tous les types de cellules car tous ne sont pas nécessaires dans la suite du projet) : à un rectangle, une ligne, ou encore un cube correspond en effet un identifiant différent.

4. **La fonction CreatePointScalarSection :**
   Input : Points (identique à FilePoints à qui la fonction SavePoints faisait appel)
   On ouvre une nouvelle section (juste en indiquant le nombre de points) où l'on va associer des valeurs à chaque point.

5. **La fonction SavePointScalar :**
   Input : Label (ce que représente les valeurs, des distances par exemple), Values (la liste des valeurs associées à chaque point)
   On écrit dans le fichier les valeurs des points précédemment déclarés. Dans notre cas, il s'agit de distance et de distance mise au cube. On pourra ensuite sur Paraview afficher ces valeurs, comme c'est par exemple le cas sur la figure ci-dessous.



Visualisation de test_file.vtk sur Paraview (qui est bien identique à exemple_cas1.vtk)

La classe tree_amr :

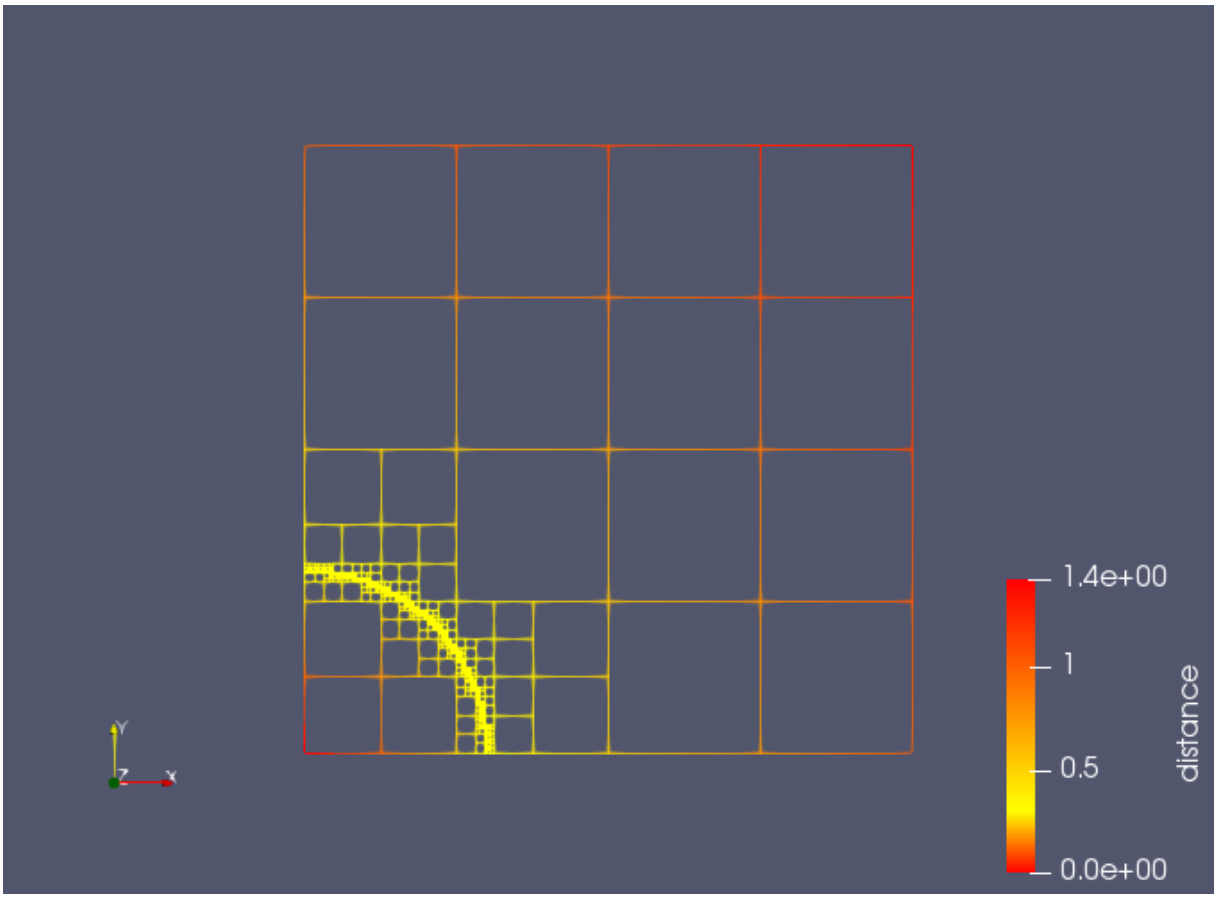
Nous allons créer un maillage qui sera très raffiné uniquement autour de l'isoligne de niveau d'une fonction, car il est inutile et très coûteux en calcul de raffiner les autres zones pour lesquelles un maillage relativement grossier suffit amplement.

1. **La fonction __init__ :**
   Input : point_0, point_1, point_2, point_3 (les quatre points du maillage), depth (la profondeur de la branche en train d'être construite), depth_max (la profondeur maximale que l'on pourra atteindre), dico (qui rassemble divers fonctions et nombres que l'on va utiliser par suite)
   La fonction permet de construire l'ensemble des branches et des feuilles du QuadTree. On définit les attributs points_0, point_1, point_2, point_3 ainsi que value_0, vlaue_1, value_2, value_3 qui sont les valeurs de la fonction en ces quatre points (que l'on calcule grâce à dico['eval_function']). La fonction dico['refine_or_not'] qui prend en entrée les valeurs des points est un booléen. S'il est vrai ou que le niveau de raffinage minimal n'a pas été atteint (donné par dico['level_min']) et si l'on ne dépasse pas le niveau de raffinage maximal, alors on appelle récursivement la fonction __init__ sur les quatre carrés qui composent le grand, tout en définissant les attributs child_up_left, child_up_right, child_down_left, child_down_right. On définit également l'attribut branch qui servira par la suite : c'est un booléen qui vaut vrai si l'on est dans une branche, faux s'il s'agit d'une feuille.

2. **La fonction Create_Mesh_And_Connectivity_List :**
   Input : List_Of_Point (la liste des points du QuadTree), Connectivity (liste des cellules)
   Cette fonction va rajouter des points et des cellules aux inputs que la classe FileVTK utilisera afin que l'on visualise notre maillage sur Paraview. Si l'objet sur lequel est appelé la fonction est une feuille, on rajoute les points à List_Of_Point et le carré formé par ces points à Connectivity. Si c'est une branche, on appelle récursivement la fonction sur les attributs child_up/down_right/left dans le but d'atteindre une feuille.

3. **La fonction Get_values :**
   Input : List_Of_Values (liste des valeurs de points étudiés)
   Cette fonction sert à récupérer les valeurs des points du maillage que l'on ajoutera ensuite sur les fichiers vtk grâce à l'objet FileVTK pour les visualiser sur Paraview. S'il s'agit d'une feuille, on rajoute à la liste les valeurs que l'on a enregistré comme attribut (dans la fonction __init__). Sinon, on appelle récursivement la fonction sur les quatre attributs child_up/down_right/left jusqu'à atteindre une feuille.

Pour vérifier que l'objet tree_amr fonctionne effectivement, on réalise le maillage d'un carré de côté 1 avec un niveau de profondeur minimale de 1 et maximale de 10. La fonction choisie (dico['eval_function']) est la distance au centre (placé en bas à gauche du carré) et on s'intéresse à l'isoligne 0,3. Nous avons rajouté dans la fonction test les lignes 186 à 189 afin d'afficher les distances sur Paraview. On obtient donc la figure suivante :



Visualisation du fichier tree_amr.vtk sur Paraview pour l'isoligne 0,3

La classe tri :

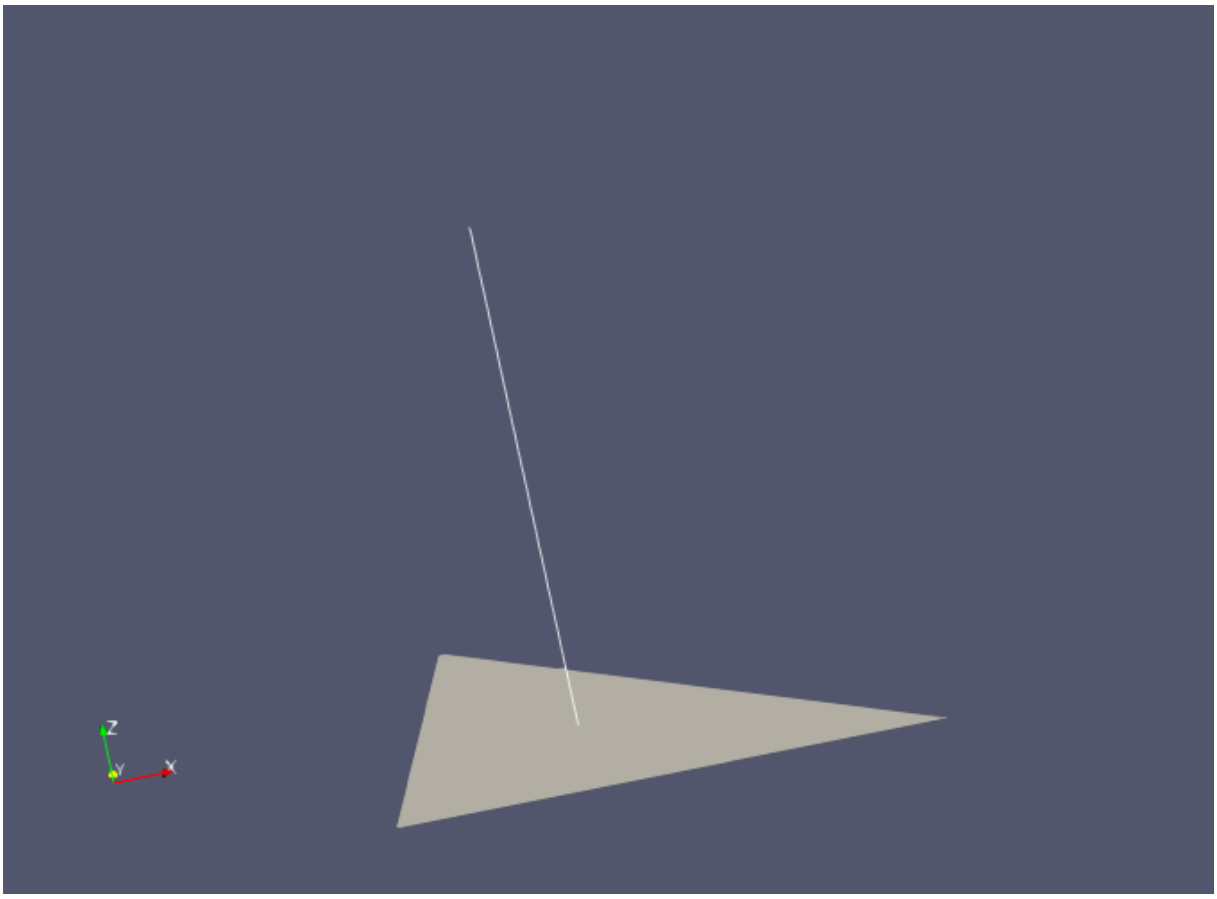
Nous allons être amenés par la suite à utiliser des tétraèdres. La classe tri va permettre de créer les triangles qui seront à la base de ces tétraèdres.

1. **La fonction __init__ :**
   Input : point 1, point 2, point3
   La fonction définit ces trois points comme des attributs. Elle crée aussi l'attribut normal (qui est la normale au triangle engendré par les trois points) : on se sert du module numpy pour calculer le produit vectoriel (np.cross) et normaliser (np.linalg.norm) très rapidement.

2. **La fonction Create_Mesh_And_Connectivity_List :**
   Input : List_Of_Point (la liste des points, repérés par leurs coordonnées cartésiennes, qui constituent les triangles), Connectivity (la liste des triangles formés à partir de ces points)
   La fonction ajoute les points qui sont en attributs à List_Of_Point et crée un nouveau triangle avec ces points dans Connectivity.

3. **La fonction Create_Tri_and_Normal_Mesh_Connectivity_List :**
   Input : List_Of_Point, Connectivity
   On reprend le travail fait par la fonction Create_Mesh_And_Connectivity_List mais on ajoute à Connectivity la normale (définie comme attribut dans la fonction __init__) ainsi que les points par lesquels passent cette normale dans List_Of_Point (on choisit le centre du triangle et le centre translaté de la normale).

4. **La fonction distance_to_a_point :**
   Input : p (un point repéré par ses coordonnées cartésiennes)
   La fonction retourne une liste constituée d'un booléen normal_side et de la distance d_min. Si normal_side est faux, la face du triangle ne pointe pas vers le point et on dit alors que d_min est infini. Si normal_side est vrai, ce que l'on peut vérifier en faisant un simple produit scalaire, alors il faut calculer d_min. On a d'abord fait une première approximation en renvoyant la distance aux sommets du triangle la plus petite parmi les trois. Ceci correspond aux lignes 360 à 362, ensuite placées en commentaire au profit d'une méthode plus précise.
   Le véritable calcul de d_min est en fait un problème assez complexe, c'est pourquoi nous nous sommes servis d'un programme sur github de Joshua Shaffer (https://gist.github.com/joshuashaffer/99d58e4ccbd37ca5d96e). On définit la fonction pointTriangleDistance qui prend le point p en entrée : on la inclue dans la classe tri et légèrement modifiée en conséquence (ligne 169, 170, 172) pour que les sommets du triangle soient bien appelés. On fait ensuite appel à cette fonction dans distance_to_a_point afin de calculer d_min de manière beaucoup plus précise que ce que l'on avait fait dans un premier temps.



Visualisation sur Paraview d'un triangle et de sa normale

La classe tetra :

Cette méthode sert à créer les tétraèdres. Elle fait appel à la classe tri pour définir les triangles qui composent le tétraèdre.

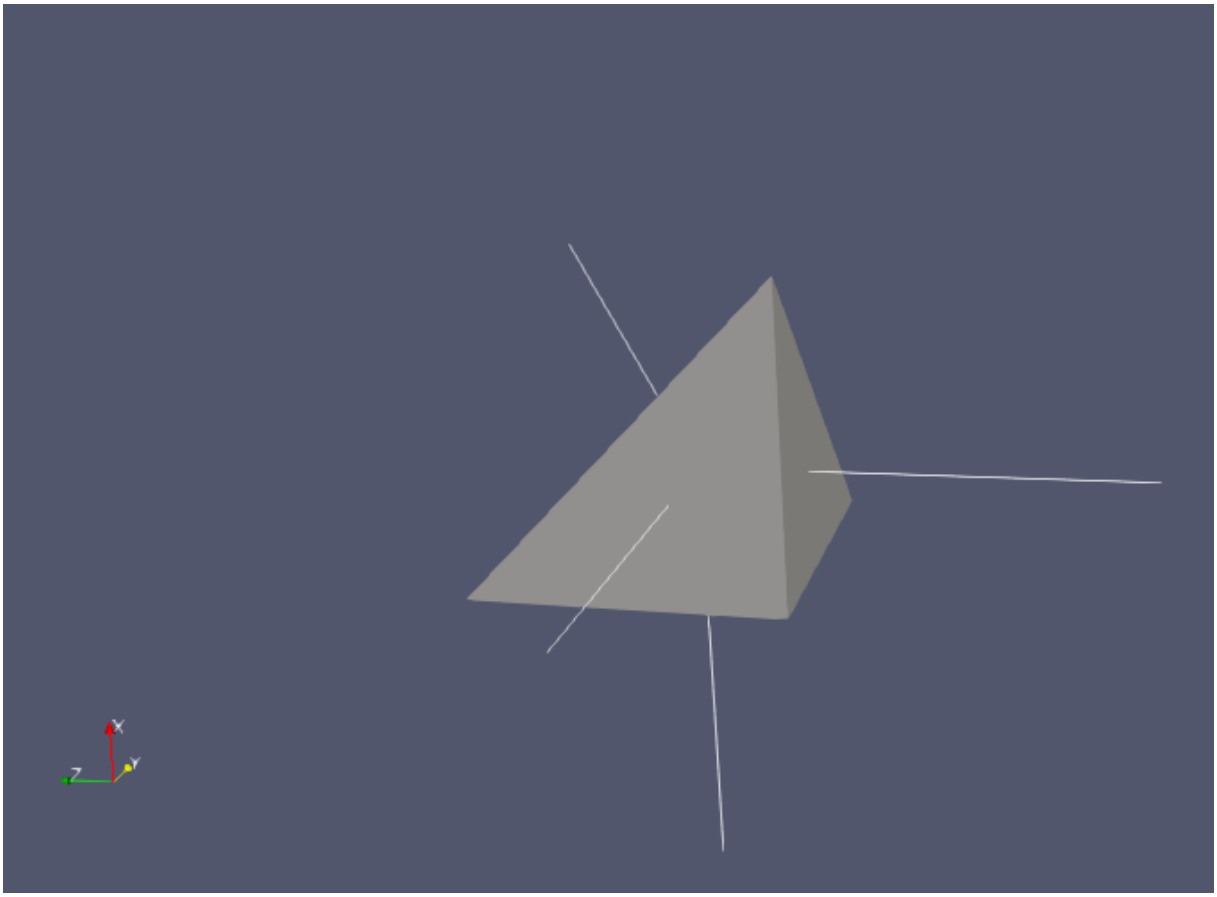
1. **La fonction __init__ :**
   Input : point1, point2, point3, point4
   On définit l'attribut points qui est une liste des quatre sommets du tétraèdre. On définit ensuite les normales en faisant des produits vectoriels, que l'on oriente vers l'extérieur grâce à des produits scalaires. On place ces normales dans une liste qui sera l'attribut faces. Enfin, on définit en attribut les quatre triangles du tétraèdre (on appelle la classe tri), ainsi que les quatre centres de ces faces.

2. **La fonction Create_Mesh_And_Connectivity_List :**
   Input : List_Of_Point, Connectivity
   On ajoute les points (qui sont en attribut) à List_Of_Point et on rajoute le tétraèdre formé par ces points à Connectivity.

3. **La fonction Create_Mesh_And_Connectivity_List_From_Tri :**
   Input : List_Of_Point, Connectivity
   Cette fonction sert à créer en plus les quatre triangles qui forment le tétraèdre. On appelle d'abord la fonction précédente pour définir le tétraèdre puis on rajoute les quatre triangles en appelant la fonction Create_Mesh_And_Connectivity_List de la classe tri sur chacun. On

rajoute ensuite les normales à Connectivity et les centres ainsi que les centres translatés des normales à List_Of_Point (nous n'avons pas utilisé la fonction Create_Mesh_And_Connectivity_List de tri car l'orientation des normales était alors quelconque).

4. **La fonction distance_to_a_point :**
   Input : p (un point repéré par ses coordonnées cartésiennes)
   En faisant quatre produits scalaires différents, on regarde si le point se trouve à l'intérieur du tétraèdre. Si c'est le cas, on renvoie le booléen outside qui sera Faux et d_min qui vaudra l'infini. Si le point se trouve à l'extérieur (outside=True), on calcule la distance minimale entre le point et les quatre faces en appelant la fonction distance_to_a_point de la classe tri (celle pour laquelle on a utilisé un code de github), puis on récupère la plus petite de ces distances qui sera d_min.



Visualisation d'un tétraèdre et de ses quatre normales sur Paraview

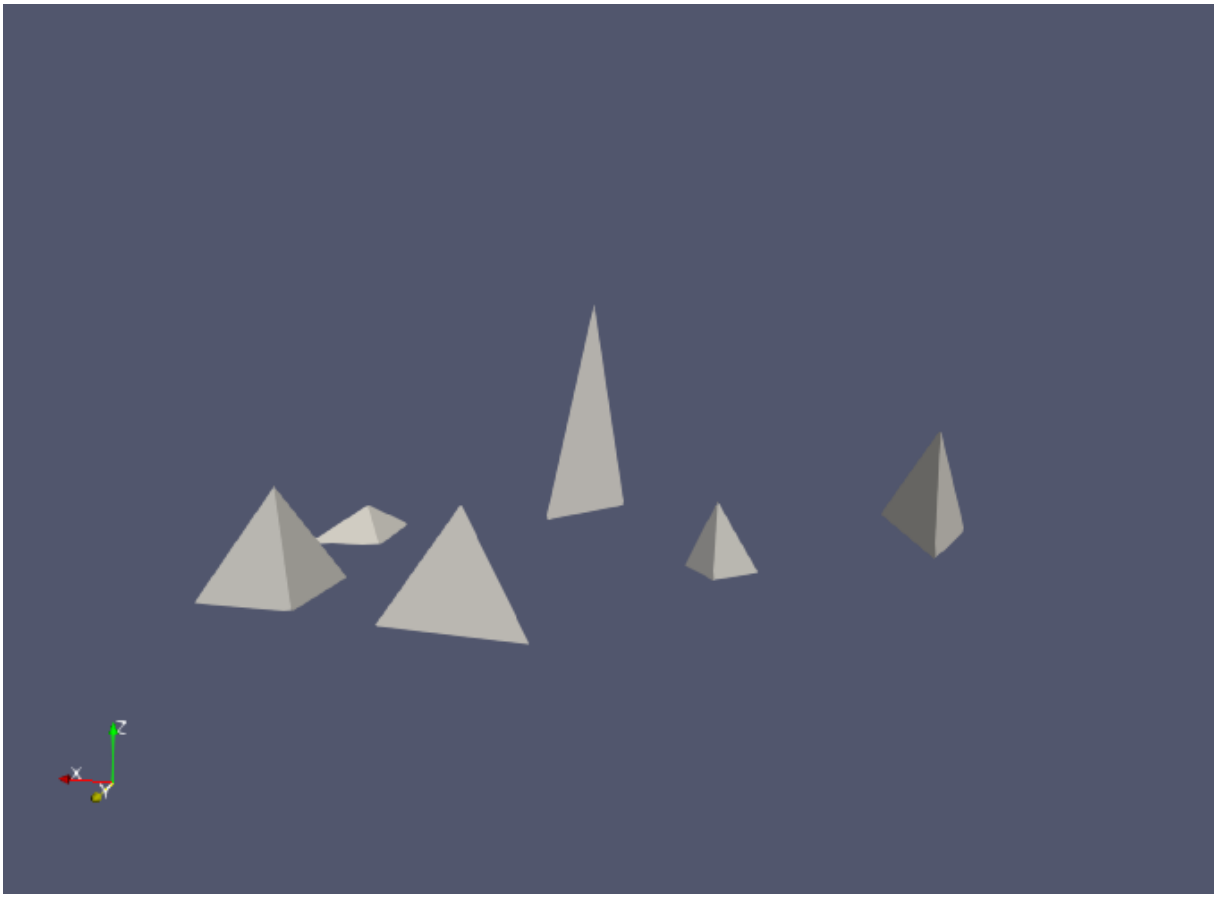
La classe mesh :

Cette classe repose principalement sur le travail fait lors des modules précédents. On crée un objet maillage, à partir duquel on pourra tracer nos tétraèdres et calculer la distance minimale d'un point au reste du maillage.

1. **La fonction __init__ :**
   Input : DefenseFile (fichier texte indiquant le nombre de tétraèdres, les coordonnées du centre de la base du tétraèdre, la longueur du côté du triangle équilatéral servant de base, la hauteur du tétraèdre et la rotation du tétraèdre autour de sa hauteur)
   On se sert des fonctions open, readline, split et np.asarray pour extraire du fichier DefenseFile les informations qui nous intéressent. Par de simples opérations géométriques (convertir la rotation en radians, multiplier par une matrice de rotation, translater par rapport au centre…), on récupère les coordonnées des sommets de chaque tétraèdre. On définit alors l'attribut List_Of_Modules qui est une liste contenant tous les tétraèdres (que l'on crée grâce à la classe tetra).

2. **La fonction Create_Mesh_And_Connectivity_List :**
   Input : List_Of_Point, Connectivity
   Pour chaque tétraèdre, on appelle la fonction Create_Mesh_And_Connectivity_From_Tri du module tetra pour compléter List_Of_Point et Connectivity.

3. **La fonction Distance_Between_a_Point_and_the_Modules :**
   Input : p (un point repéré par ses coordonnées cartésiennes)
   La fonction renvoie le booléen outside qui vaut True si le point est à l'extérieur de tous les tétraèdres et d_min qui est une liste contenant les distances minimales entre le point et les tétraèdres. Pour cela, on utilise simplement la fonction distance_to_a_point sur le point et chacun des tétraèdres (en attribut dans List_Of_Modules).



Visualisation sur Paraview des tétraèdres de DefenseZone

Le module 6 :

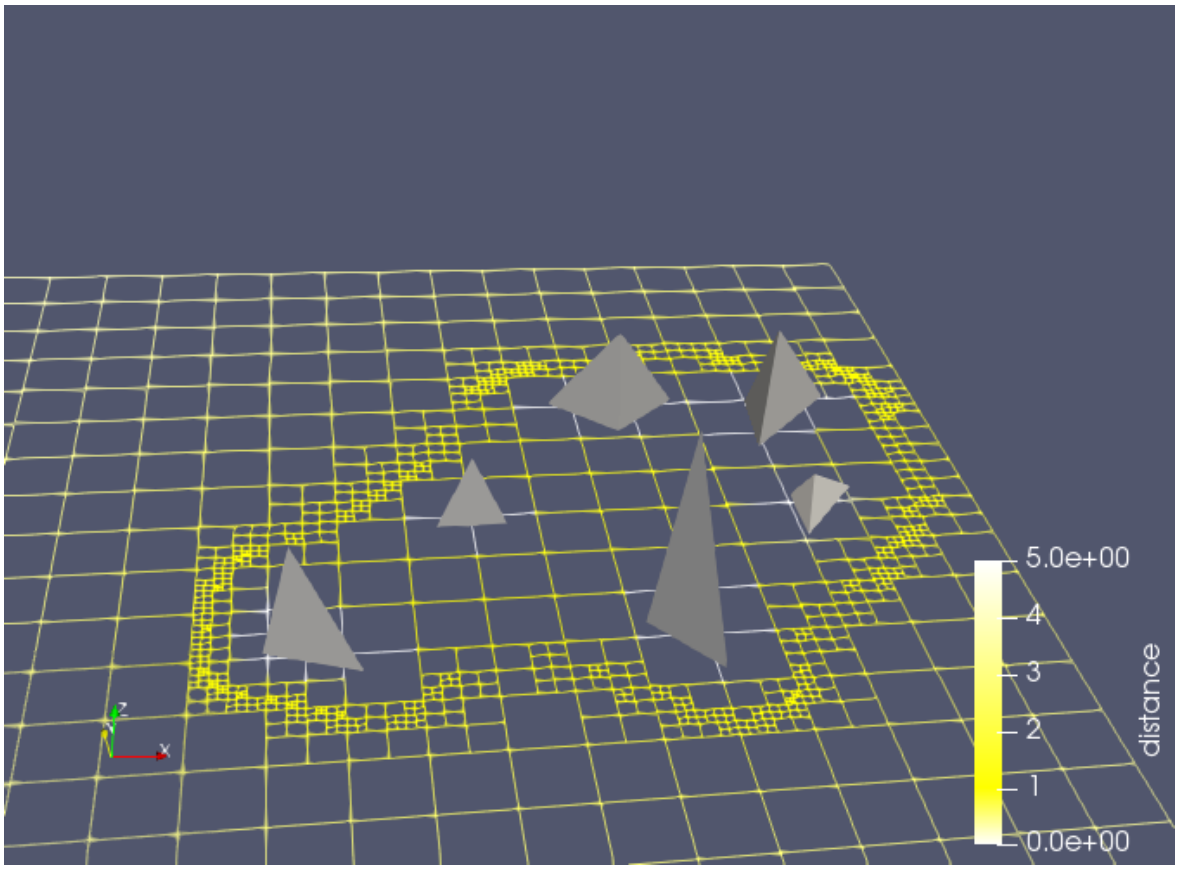
Pour ce script, on assemble et adapte tout ce qui nous est utile dans les autres scripts pour aboutir un résultat final attendu.

1. **La fonction compute_value :**
   Input : p (un point), dico
   Cette fonction renvoie S, la somme des 1/d où d est la distance minimale entre un point et un tétraèdre. Ces distances sont récupérées en appelant la fonction Distance_Between_a_Point_and_the_Modules de la classe mesh sur le fichier defensezone. Si le point est dans un tétraèdre, alors cette distance S vaut l'infini.

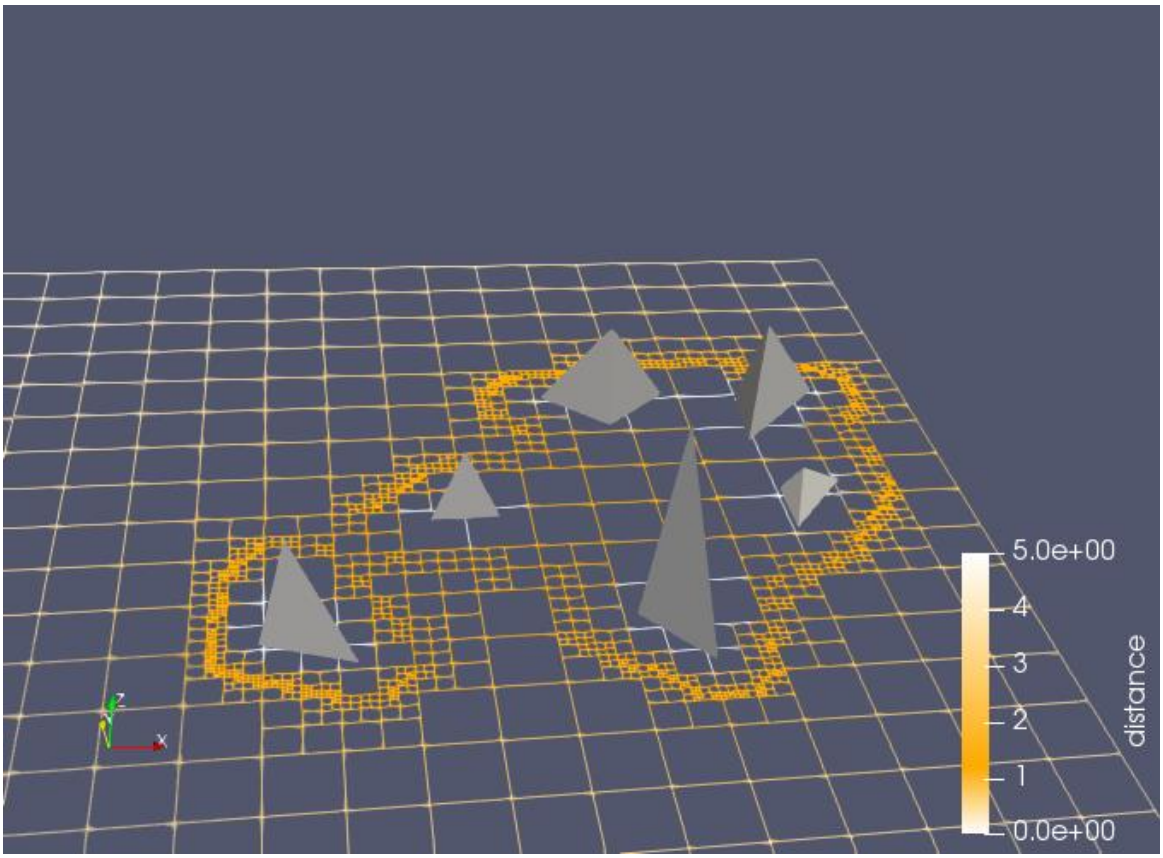
2. **La fonction test_function :**
   Input : r0, r1, r2, r3 (ce sont les valeurs des points des sommets du carré étudié), dico (en particulier dico['r_target'] qui sera l'isoligne recherchée).
   On reprend ici la fonction qui avait été codée dans un module précédent. On rajoute cependant la ligne 36 : la ligne de niveau est ainsi définie à 10 cm près.

3. **La fonction test (sans input) :**
   On définit les quatre points du maillage initial (on choisit une hauteur z = 0.01 m) ainsi que la profondeur maximale. On choisit comme fonction d'évaluation compute_value, comme fonction de choix pour le raffinage test_function. Quant à dico['r_target'], il vaudra successivement 1, 1.2 et 1.5 afin de visualiser trois isolignes différentes. On crée ensuite le QuadTree grâce à la classe tree_amr, on sauvegarde les points, les cellules et les valeurs grâce à la classe FileVTK.

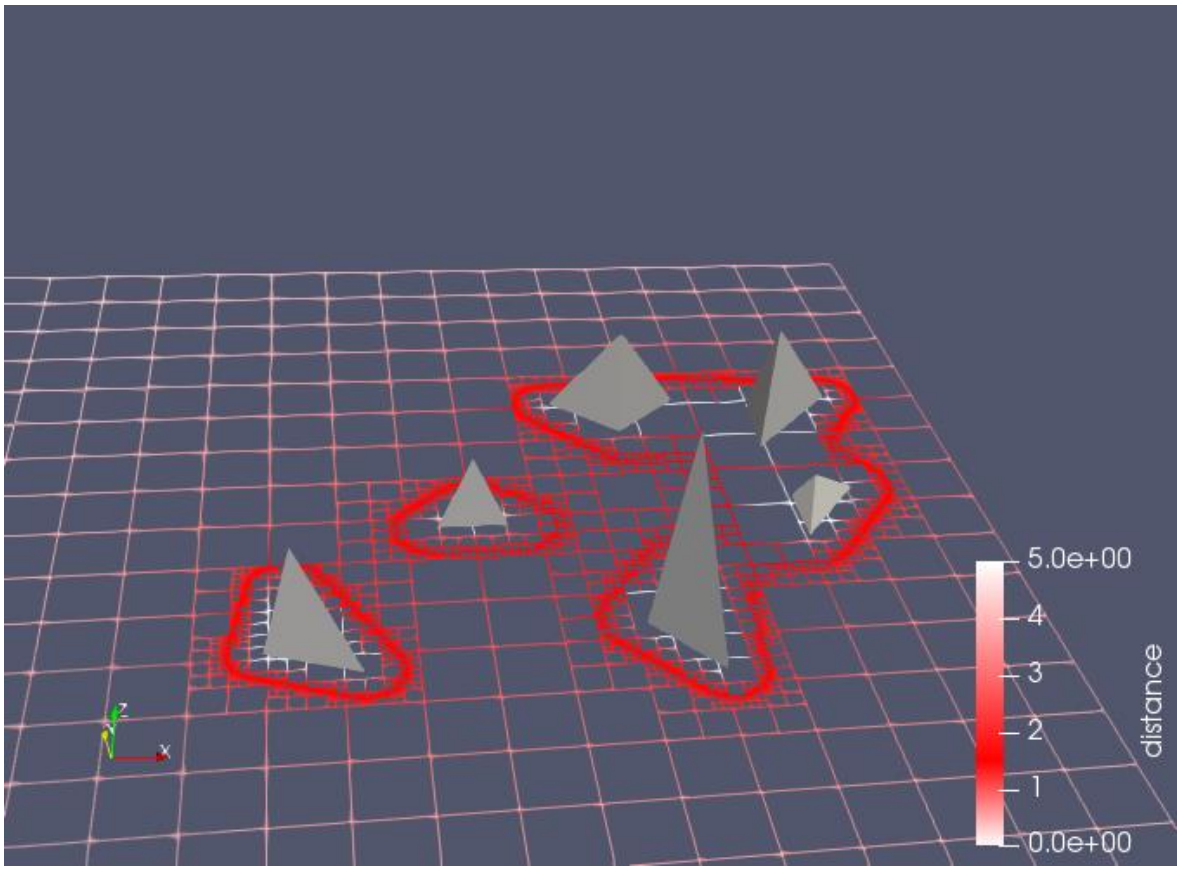
On affiche ensuite les isolignes sur Paraview : grâce au travail fait sur Get_value, on peut visualiser la distance en modifiant l'échelle de couleur.



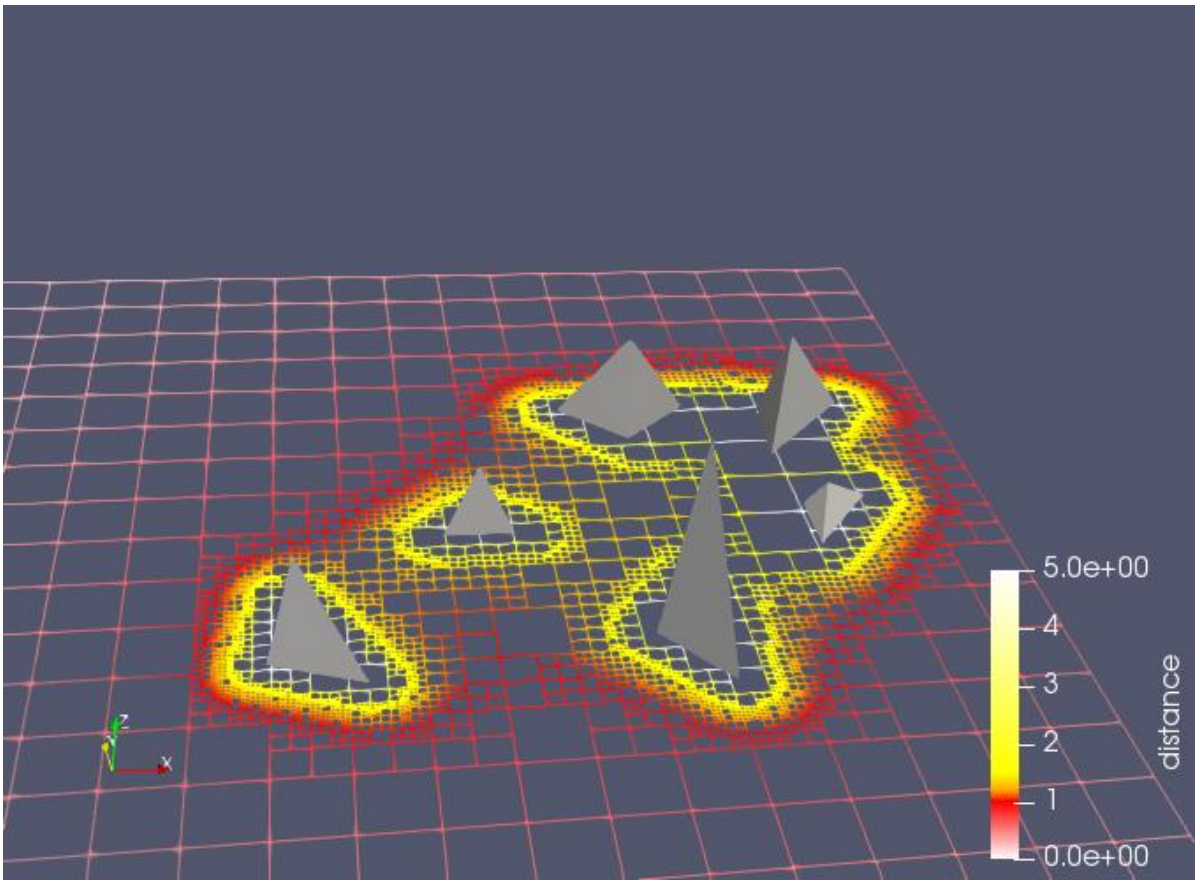
Visualisation sur Paraview de l'isoligne de niveau 1



Visualisation sur Paraview de l'isoligne de niveau 1.2



Visualisation sur Paraview de l'isoligne de niveau 1.5



Visualisation sur Paraview des trois isolignes de niveau 1, 1.2 et 1.5