Rapport Projet CAO

Interpolation par élément fini de Powell et Sabin pour la représentation d’une surface de classe C1

Introduction :

Le but de ce projet a été de développer un algorithme capable d’interpoler une fonction donnée sur un domaine D composé d’éléments finis de Powell et Sabin, où .

Pour cela on dispose d’un fichier *ps.pts* contenant les coordonnées de N points dont l’enveloppe convexe est le domaine D.

L’algorithme doit renvoyer un fichier de résultats *PS.RES* que l’on détaillera par la suite.

1. Génération de la triangulation

La première phase a été de générer la triangulation du domaine D à l’aide du script Matlab *quest1.m*. Dans ce script on charge le fichier *ps.pts* et on crée une triangulation du domaine avec la fonction « delaunay ». On crée par la suite un fichier *listri.dat* contenant les trois sommets de chaque triangle présent dans le domaine après triangulation de ce dernier. On crée également un fichier *points.pts* qui renvoie les coordonnées de tous les points de la triangulation rangés dans l’ordre croissant. C’est à partir de ces deux fichiers que l’algorithme, implémenté en C++, pourra calculer l’interpolant pour une fonction donnée dans le domaine D.

1. Structure des données

La première étape de l’implémentation en C++ à donc été de récupérer les données présentes dans les fichiers *listri.dat* et *point.pts.* Nous avons fait le choix de créer une classe « Point » permettant de contenir les coordonnées cartésiennes et barycentrique d’un point.

1. Classe Point

L’implémentation de la classe Point a été faite dans les fichiers *Point.cpp* et *Point.h*. Un Point est composé de deux coordonnées cartésiennes et de trois coordonnées barycentriques non définis lorsqu’on ne ce trouve pas dans un triangle en particulier.

|  |
| --- |
| class Point{  private:  // Données  double X,Y; // Coordonnées cartésiennes  double w1,w2,w3; //coordonnées barycentriques  public:  // constructeur & destructeur  Point() { X = Y = -1.; }  Point( double xi, double yi);  ~Point();  // fonctions  void affiche(void)const;  void attrib\_coord(double x, double y);  void attrib\_bary(double c1, double c2, double c3);  void getCart(double &x,double &y);  void getBary(double &c1, double &c2, double &c3);  }; |

On observe deux constructeurs qui permettent de définir les coordonnées cartésiennes d’un point. La classe comporte les méthodes suivantes, implémentés dans *Point.cpp*:

* affiche : permet d’afficher les coordonnées cartésiennes et barycentriques du Point.
* attrib\_coord : permet d’attribuer les coordonnées cartésiennes du Point à partir de deux doubles pris en entrée.
* attrib\_bary : permet d’attribuer les coordonnées barycentriques du Point, dans un triangle choisi au préalable, à partir de trois doubles pris en entrée.
* getCart : permet de renvoyer les coordonnées cartésiennes du Point à l’adresse de deux doubles pris en entrée de la fonction.
* getBary : permet de renvoyer les coordonnées barycentriques du Point àl’adresse de deux doubles pris en entrées dans la fonction.

1. Matrice

Pour pouvoir gérer les données nous avons également fait le choix de créer des matrices allouées dynamiquement. Toutes fonctions relatives à des matrices à donc été implémenté dans le fichier *Matrice.h* contenant les fonctions templates suivantes :

* CreateMat : permet de créer une matrice de taille , où nrow et ncol sont deux entiers pris en entrée.
* AfficheMat : permet d’afficher une matrice de taille .
* AfficheVect : permet d’afficher un vecteur de taille
* FreeMat : permet de désallouer la mémoire prise lors de la création d’une matrice

1. Récupération des données

La lecture des données du fichier *point.pts* se fait dans la fonction LecPoints implémenté dans *Point.cpp*. Dans cette fonction on lit les coordonnées de chaque Point et on les stocke en sortie de fonction, dans un vecteur de NbPts Points, où NbPts est le nombre de points dans la triangulation. Cette fonction est appelée dans le fichier *main.cpp*, ainsi on crée le vecteur de Points ListPoints dans le fichier principal.

La lecture des données du fichier *listri.dat* se fait quant à elle dans la fonction lectTriangles implémenté dans *Triangles.cpp.* Cette fonction renvoie une matrice d’entiers de taille , où NbTri est le nombre de triangle dans la triangulation. Cette matrice comporte les numéros des 3 sommets de chaque triangle. On nomme cette dernière NT dans le fichier main.cpp.

1. Création des matrices de données

Par la suite nous avons créé une matrice NTV de taille , elle contient, pour chaque triangle, le numéro des trois triangles voisins. Si le triangle ne possède pas de voisins sur l’un de ses cotés alors on a affecté la valeur -1. Pour créer cette matrice on a utilisé la fonction initNTV situé dans le fichier *Triangle.cpp.* Cette fonction utilise elle-même la fonction triangleVoisin qui permet de trouver le triangle voisin d’un coté donnée.

Nous avons également créé un vecteur de Points comportant les points Ω, centre de chaque triangle du domaine. Le vecteur Omega est trié de la même façon que la matrice NT, et il est initialisé grâce à la fonction initOmega, implémenté dans le fichier *Triangle.cpp*. Cette fonction permet de trouver les coordonnées cartésiennes du centre Ω de chaque triangle à partir des coordonnées barycentriques de Ω pour chaque triangle. Le vecteur Omega contient donc le centre du cercle inscrit de chaque triangle, et leur coordonnées cartésiennes et barycentriques sont initialisés.

La dernière matrice créée est la matrice de Points NM de taille , elle contient les Points Mi de chaque triangle. Les coordonnées cartésiennes des Points Mi, pour un triangle k sont calculés comme étant l’intersection du côté opposé à Ai avec le segment Ω Ω’, où Ω’ est le centre du cercle inscrit du triangle voisins au triangle k passant par les sommets Aj et Ak. Si le triangle n’a pas de voisin sur un de ces côtés alors on place le point Mi au centre du côté opposé au sommet Ai. Ce calcul est fait dans la fonction initNM implémenté dans le fichier *Triangle.cpp*. Les coordonnées barycentriques des Points Mi sont quant à elles calculés dans la fonction CoordBaryMi situé dans le même fichier que précédemment.

Pour finir on crée un vecteur SMT à 3 dimensions de taille . Ce vecteur contient les 3 sommets de chacun des 6 micro-triangles contenu dans chaque triangle. On initialise ce vecteur grâce à la fonction ComputeAllSMT implémenté dans *Triangles.cpp.*

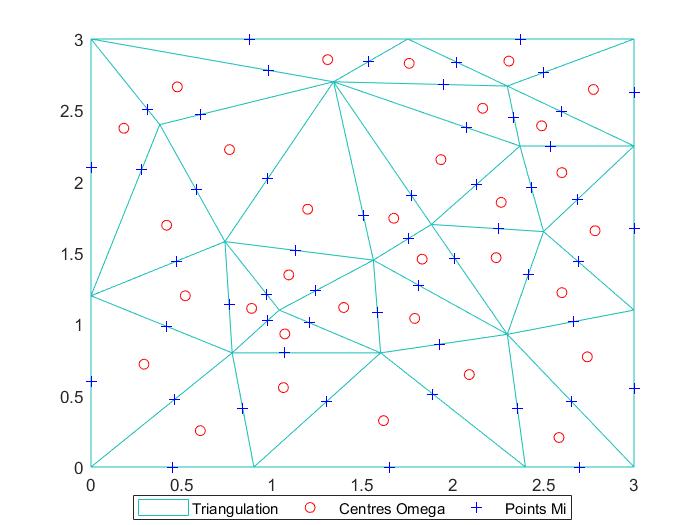


Figure 1: Triangulation du domaine [0,3]\*[0,3]

1. Calcul de l’interpolation dans le domaine D

La seconde étape de l’implémentation en C++ à été de pouvoir calculer la surface d’interpolation pour une fonction donnée. Ainsi pour chaque fonction il a fallu calculer les coefficients de chaque triangle, être capable de localiser un point dans la triangulation puis dans la micro-triangulation du triangle en question, et enfin calculer l’interpolant pour ce point.

* 1. Fonctions évaluées

Pour cette étude nous avons utilisés deux fonctions  et . Ces deux fonctions ont été implémentés dans le fichier *Fonction\_F\_G.cpp*. On affecte le numéro de fonction 1 pour la fonction f et 2 pour la fonction g.

Ce fichier contient les fonctions suivantes :

* f : permet de prendre un Point en entrée et de renvoyer la valeur de la fonction en ce Point. Le numéro de la fonction choisi est également demandé en entrée pour déterminer si on évalue le Point avec la fonction f ou g.
* fpx : permet de prendre un Point en entrée et de renvoyer la dérivée par rapport à x de la fonction dont le numéro a été requis en argument d’entrée.
* fpx : permet de prendre un Point en entrée et de renvoyer la dérivée par rapport à y de la fonction dont le numéro a été requis en argument d’entrée.
* gradf :permet de prendre un Point en entrée et de renvoyer un objet de type Point contenant pour coordonnées cartésienne (fpx,fpy) détérminé à partir du numéro de fonction demandé en argument d’entrée.

L’avantage de cette méthode est que si l’on souhaite évaluer l’interpolant pour une troisième fonction on a juste à rajouter le cas où le numéro de fonction correspond à 3 pour chacune des fonctions présentées ci-dessus.

* 1. Localisation d’un point

Pour pouvoir évaluer l’interpolant par la suite nous avons besoin de pouvoir situer un Point dans la triangulation et plus précisément dans un micro-triangle de la triangulation.

Pour situer un Point dans la triangulation on utilise la fonction LocatePointTriangle. Cette dernière permet de calculer les coordonnées barycentriques du Point dans le triangle 0. Si l’une des coordonnées est négative alors on dit que le Point est dans ce triangle ci. Sinon on fait la même chose sur le triangle suivant et ainsi de suite jusqu’à tomber sur le bon triangle.

Une fois le triangle trouvé il reste à savoir dans quel micro-triangle le Point se trouve. Pour cela on utilise la fonction LocatePointMicroTriangle qui utilise la même méthode que la fonction précédente mais appliqué cette fois aux micro-triangles.

* 1. Calcul des coefficients et de l’interpolant

Pour chaque triangle on souhaite calculer les 19 coefficients qui serons utilisé par la suite dans le calcul de l’interpolant. Pour cela on utilise la fonction ComputeAllCoeffs qui permet de renvoyer une matrice de taille , que l’on nommera AllCoeff dans le fichier *main.cpp*. Cette fonction prend en argument d’entrée le numéro de la fonction à évaluer pour savoir si on étudie la fonction f ou g.

Après avoir calculé les coefficients on peut calculer l’interpolant d’un point quelconque dans le domaine D. Ce calcul se fait avec la fonction evalInterpolant. Cette dernière fait appel à la fonction LocatePointTriangle qui permet de localiser dans quel triangle est le point sur le domaine D. Puis elle fait appel à la fonction LocatePointMicroTriangle qui permet de localiser dans quel micro-triangle est le point. Et enfin elle fait appel à la fonction foncInterpolant qui permet de calculer l’interpolant pour un micro-triangle donné.

La fonction evalInterpolant renvoie donc un double correspondant à l’interpolation du Point donnée en argument pour la fonction dont le numéro à été également donné en argument.

1. Résultats
   1. Visualisation de la surface interpolée

Pour pouvoir visualiser la surface interpolée on a créé une grille de 100 Points par 100 sur le domaine D. Et on a interpolé chacun de ces Points avec la fonction f et g. La fonction FichierSurface permet de générer cette grille et elle écrit dans un fichier la valeur de l’interpolant pour chacun des Points de cette grille.

* 1. Fichier PS.RES

Comme énoncé dans l’introduction l’algorithme doit renvoyer un fichier *PS.RES*, ce dernier est généré de la façon suivante :

* Dans un premier temps on rentre la liste des triangles du domaine avec les sommets de chaque triangle et les coordonnées cartésiennes des centres du cercle inscrit Ω de chaque triangle. Ceci est fait grâce à la fonction results\_ListTriangles.
* On rentre par la suite les coordonnées cartésiennes des Mi de chaque triangle. Ceci est fait grâce à la fonction results\_ListPoints.
* Puis pour chaque fonction étudiée, c’est-à-dire f et g :
  + On rentre la liste de tous les points de la triangulation avec leurs coordonnées cartésiennes. Ainsi que la valeur d’évaluation du chaque point par la fonction (f ou g), et par ses dérivées en x et en y. Ceci est fait grâce à la fonction results\_ValFonc.
  + Puis on rentre la valeur de l’interpolant aux points (2.5,0.8), (0.2,1.1) et (2.9,2.5). Ceci est fait grâce à la fonction resuls\_Interpol.
  + Et enfin on affiche le minimum et le maximum sur l’ensemble des points utilisés pour la visualisation de l’interpolation sur le domaine. Cette dernière partie est réalisé par la fonction results\_Erreur.