Interpolation de Sibson

Tong Zhao, Pierre Boyeau

Mai 2017

Introduction

Les techniques usuelles d'interpolation mettent parfois en avant des problèmes de régulaté pour certains fonctions.
L'interpolation de Sibson permet souvent d'éviter ces écueils en permettant une interpolation plus fine, basée sur la notion de voisins naturels.

1 Principe de la méthode

1.1 Coordonnées de Sibson

On se dote d'un ensemble de points d'entraînement $(\mathbf{p}_i)_{i\in\{1,\dots l\}}$, dont on calcule le diagramme de Voronoi. On note V_i^0 les zones d'influences associées.
Les **coordonnées de Sibson** d'un point $\mathbf{x}$ correspondent aux quantités $\lambda_i(\mathbf{x})$ telles que :

$$\mathbf{x} = \sum_{i=1}^{l} \lambda_i(\mathbf{x})\mathbf{p}_i$$

Pour calculer les valeurs des $\lambda_i(\mathbf{x})$, on suit le protocole suivant:

- On calcule le diagramme de Voronoi de $(\mathbf{p}_1 \dots \mathbf{p}_l)$ auquel on a ajouté $\mathbf{x}$. On note V_x la zone d'influence associée à x
- $\lambda_i(\mathbf{x})$ correspond au rapport $\frac{\pi_i(\mathbf{x})}{\pi(\mathbf{x})}$ où $\pi_i(\mathbf{x}) = \mathcal{A}(V_i^0 \cap V_x)$ et $\pi(\mathbf{x}) = \mathcal{A}(V_x)$.

  Plus qualitativement, la coordonnée $\lambda_i(\mathbf{x})$ correspond à l'aire de la zone d'influence de i mangée par x quand on introduit celui-ci dans le diagramme de Voronoi.

La figure suivante permet de visualiser les coordonnées de Sibson.

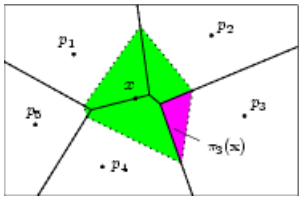

Figure 1: Coordonnées de Sibson

1.2 Interpolation

Une fois les coordonnées de Sibson de $\mathbf{x}$ il est très facile d'interpoler l'intensité de l'image au point $\mathbf{x}$ à partir de l'ensemble d'entraînement $(\mathbf{p}_i)_{i\in\{1,\dots l\}}$ par la formule suivante:

$$\hat{I}(\mathbf{x}) = \sum_{i=1}^{l} \lambda_i(\mathbf{x}) I(\mathbf{p}_i)$$

1.3 Propriétés

1. $x = \sum_{i=1}^{n} \lambda_i(x) p_i$
2. $\forall i, j \leq n, \lambda_i(p_j) = \delta_{ij}$, où δ_{ij} est le symbole de Kronecker.
3. $\sum_{i=1}^{n} \lambda_i(x) = 1$

2 Implémentation

En C++, nous avons implémenté en 2D un programme qui interpole selon différents protocoles la couleur d'une image, en nous appuyant sur la bibliothèque *CGAL*. On propose aussi un protocole pour estimer et gérer l'erreur du résultat.
Les algorithmes de l'interpolation réalisés sont comme suit:

- Points d'entraînement choisis aléatoirement
- A erreur d'interpolation imposée, interpolation par ajout aléatoire de points
- A erreur d'interpolation imposée, interpolation par ajout hiérarchique de points
- A nombre de points fixés, minimisation du nombre de points d'interpolation nécessaires (optimisation).

On travaille avec une image couleur dans toute la suite de l'exercice. On considèrera l'image suivante:

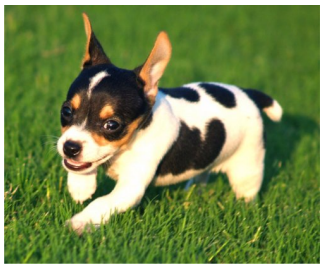

Figure 2: Image de référence de taille 500×410

2.1 Le protocole de l'erreur

Pour estimer l'erreur, on suit l'idée de l'erreur moyenne d'approximatio(MAE). Si l'on note $p_1, p_2, \ldots, p_n$ les points à interpoler, chacun entre eux a trois canaux de couleurs - rouge, vert, bleu de valeurs correspondantes $R(p_i)$, $G(p_i)$, $B(p_i)$ $\forall p \in \{1, 2, \ldots, n\}$. En notant $\hat{R}(p_i)$, $\hat{G}(p_i)$, $\hat{B}(p_i)$ les valeurs interpolées pour les trois canaux, la MAE de l'image est calculée par la formule suivante:

$$MAE = \frac{\sum_{i=1}^{n} (|\hat{R}(p_i) - R(p_i)| + |\hat{G}(p_i) - G(p_i)| + |\hat{B}(p_i) - B(p_i)|)}{3n}$$

2.2 Interpolation avec points d'entraînement choisis aléatoirement

La première démarche que nous avons adopté a été de choisir aléatoirement les points d'entraînement utilisés pour l'interpolation.
Etant donné un nombre de points m fixe, on démarre l'initialisation avec les 4 points aux coins de l'image, qui assure que tous les points à insérer sont à l'intérieur du convexe. En effet, si ce n'est pas le cas, on peut avoir des problèmes numériques pour calculer les $\lambda_i(\mathbf{x})$ (le dénominateur de cette quantité est infini!).
Sont ensuite ajoutés $m - 4$ à l'ensemble d'apprentissage.
Pour la phase de l'interpolation, on applique simplement l'algorithme suivant:

Algorithm 1 Algorithme de calcul d'interpolée de Sibson

Require: Liste L des points d'entraînement initialisée aléatoirement

```
for x = 1 : w do
    for y = 1 : h do
        Calcul de l'interpolée de Sibson I_s(x, y) à partir de L
    end for
end for
```

A l'aide de cet algorithme, on obtient les résultats suivants :

Figure 3: m = 2500

Figure 4: m = 10000

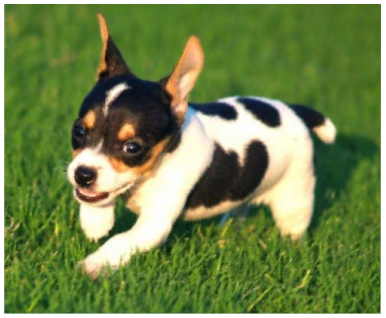

Figure 5: m = 100000

L'image du haut correspond à l'image interpolée, celle du dessous à l'image de l'erreur. On remarque que très intuitivement, plus le nombre de points d'entraînement augmente, plus l'erreur d'interpolation diminue (i.e. s'assombrit), et plus l'image gagne en netteté. C'est très clair sur le pelage du chien ou sur son oeil, qui devient très net pour 100000 points.

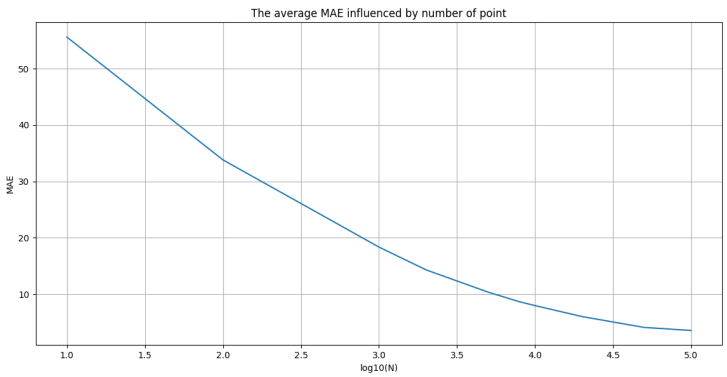

Figure 6: MAE en fonction du nombre de points d'entraînement

On trace la graphe de MAE en fonction du nombre de points d'entraînement. Sur le graphe ci-dessus, cette erreur diminue progressivement quand la taille de l'ensemble d'entraînement augmente. C'est cohérent.

2.3 Interpolation par ajout aléatoire de points

On veut affiner le processus de création d'image interpolée. Pour cela, on va s'appuyer sur l'erreur d'interpolation. L'idée de l'algorithme est d'ajouter un par un des points à l'ensemble d'entraînement aléatoirement jusqu'à satisfaire une condition sur l'erreur d'interpolation(MAE).
On suit pour cela l'algorithme suivant:

Algorithm 2 Algorithme de gestion d'erreur itératif

Require: Liste L des points d'entraînement initialisée (4 points au coin), intensité I de l'image étudiée

1: `erreur` $= \infty$
2: **while** une erreur quelconque sur les trois canaux $>$ seuil **do**
3: ajout un point à L aléatoirement
4: **for** x, y **do**
5: Calcul de l'interpolée de Sibson $I_s(x, y)$ à partir de L
6: **end for**
7: `erreur`$(x, y) = |I_s(x, y) - I(x, y)|$
8: `erreur` $=$ `MOYENNE`(`erreur`(x, y))
9: **end while**

Néanmoins, cet algorithme a un problème pratique souligné par la figure suivante:

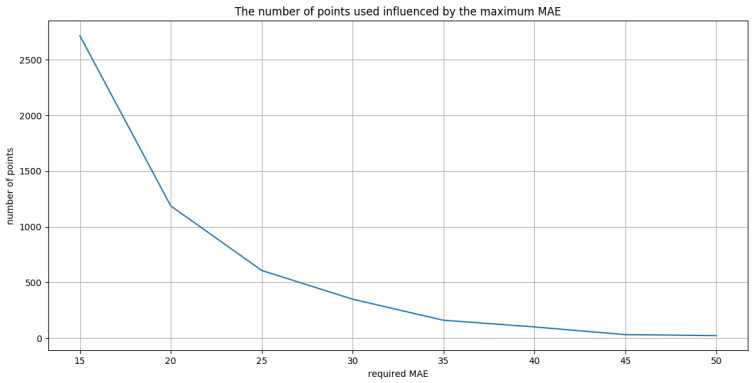

Figure 7: MAE en fonction du nombre de points d'entraînement

En effet, on s'aperçoit que pour des seuils exigeants, cette méthode est très lente à converger, et donc totalement inefficace.

2.4 Interpolation par ajout hiérarchique de points

C'est pour cette raison que nous avons considéré cette méthode. L'idée est d'ajouter non plus un point après l'autre, mais d'ajouter ceux-ci **hiérarchiquement** et intelligemment à l'ensemble de test, en suivant l'algorithme suivant:

Algorithm 3 Algorithme de gestion d'erreur hiérarchique

Require: Liste L des points d'entraînement initialisée (4 points au coin), intensité I de l'image étudiée

1: `erreur` $= \infty$
2: **while** erreur $>$ erreur seuil **do**
3: choix aléatoirement un point p
4: **for** l_i in L **do**
5: Ajout du centre de segment pl_i à L
6: **end for**
7: **for** x, y **do**
8: Calcul de l'interpolée de Sibson $I_s(x, y)$ à partir de L
9: **end for**
10: `erreur`$(x, y) = |I_s(x, y) - I(x, y)|$
11: `erreur` $=$ `MOYENNE`(`erreur`(x, y))
12: **end while**

Dans cet algorithme, la méthode d'ajout de points proposé est inspiré par l'idée suivante: pendant chaque itération, nous voulions doubler le nombre de points dans l'ensemble d'apprentissage en ajoutant les points relatifs aux points dans l'ensemble.
Ainsi, cet algorithme est utilisable en pratique pour obtenir une interpolation satisfaisante. Triviale dans le cas d'une image, elle tirerait tout son intérêt dans des problèmes plus compliqués d'interpolation, en grande dimension par exemple.

2.5 Optimisation du nombre de points d'interpolation nécessaires

Enfin, pour comprendre tout le potentiel de l'interpolation de Sibson, nous avons tenté de mettre en lumière l'importance de la position des points d'entraînement. Pour cela, à un nombre de points fixé, nous avons optimisé la position de ces points afin de minimiser l'erreur totale d'interpolation, en suivant l'algorithme suivant:

Algorithm 4 Algorithme de gestion d'erreur nécessaires

Require: Liste L des n points d'entraînement initialisée, intensité I de l'image étudiée

while Insatisfaction de la condition **do**
  for x, y **do**
    Calcul de l'interpolée de Sibson $I_s(x, y)$ à partir de L
    Calcul et sauvegarde de l'erreur en (x,y)
  end for
  Le tri de l'erreur par l'order décroissante
  for i $= 1 : int(\frac{n}{100}) - 4$ **do**
    Le remplacement du premier point dans la liste par le point dont l'erreur est le plus grand
  end for
  Le remplacement des premiers 4 points dans la liste par les 4 points au coins de l'image
  erreur = MOYENNE($|I_s(x, y) - I(x, y)|$)
end while

Pendant chaque itération, on remplace 1% de points. Pour assurer que tous les points à interpoler sont à l'interieur du diagramme de Voronoi, on met toujours les 4 points au coins à la tête de la liste.
Plus précisément, en s'inspirant de techniques de early-stop, on utilise l'algorithme suivant pour déterminer la condition d'arrêt:

Algorithm 5 Algorithme de la condition de l'arrêt

Require: Un compteur c, le meilleur résultat obtenu $e^\star$ et le résultat dans cette itération e

if $e < e^\star$ **then**
  $e \to e^\star$
  $c = 0$
else if $c < 10$ **then**
  $c + 1 \to c$
else
  $e^\star \to e$
  stop
  L'affichage de courbe de l'erreur en python automatiquement
end if

La différence entre un ensemble d'entraînement aléatoire et un ensemble optimisé est frappante. Sur le graphe suivant, on trace les interpolées (ainsi que les erreurs associées) des 2 cas pour $n = 10000$ points d'entraînement:

Figure 8: Image après interpolation aléatoire

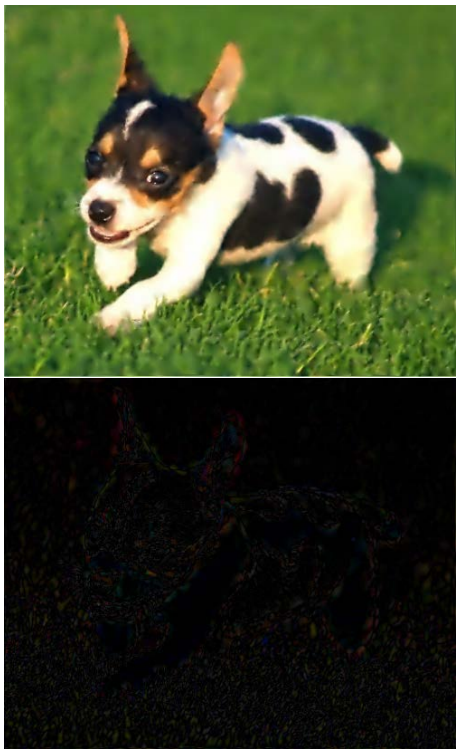

Figure 9: Image après interpolation optimisée

Ainsi, on remarque que l'image de droite est très bonne, et est presque indiscernable à l'oeil nu de l'interpolée obtenue plus haut pour 100000 points d'entraînement, soit 10 fois plus!
On affiche la courbe de l'erreur sur l'ensemble de points à interpoler dans le cas où $n = 1000$ et $n = 10000$ dans la graphe suivante:

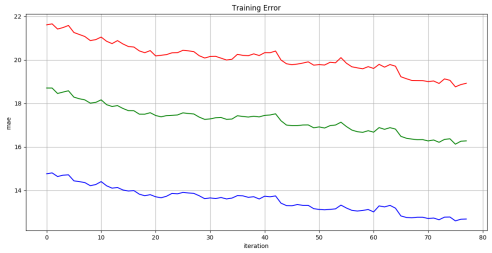

Figure 10: Courbe de l'erreur (n=1000)

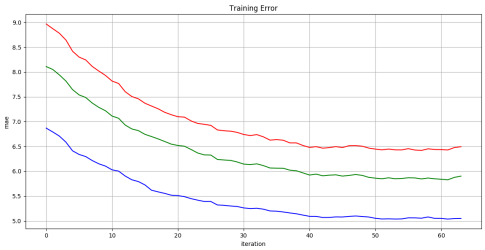

Figure 11: Courbe de l'erreur (n=10000)

La graphe à droite démarre par $MAE = 8$ et finit par $MAE = 5.5$. Le proccessus d'optimisation prend 77 itérations et le temps de calcul est $t = 506.301s$.
On s'aperçoit aussi la courbe à gauche est moins jolie que celle à droite à cause le faible nombre de points n. On en déduit que l'algorithme de l'optimisation marche mieux quand le nombre de points n est assez grand i.e. quand il a capacité de représenter toute l'image.
Maintenant, on teste notre algorithme sur une image plus difficile avec les bords clairs. La taille de l'image est 542×612.
On choisit $n = 10000$ et on fait la interpolation par ajout de points aléatoirement et par interpolation optimisée.

Figure 12: Image originale

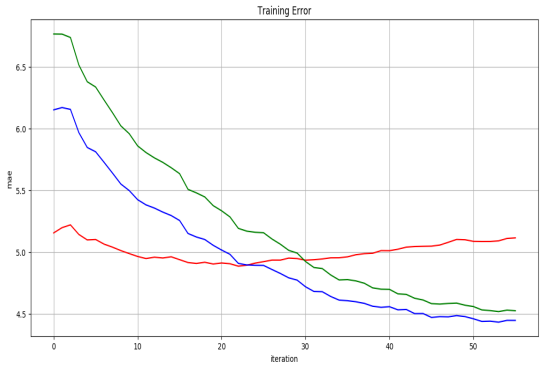

Figure 13: Courbe d'erreur

Figure 14: Image après interpolation aléatoire

Figure 15: Image après interpolation optimisée

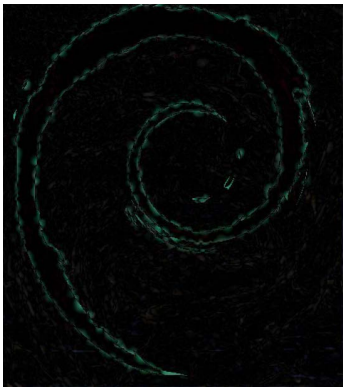

Figure 16: Image de différence après interpolation aléatoire

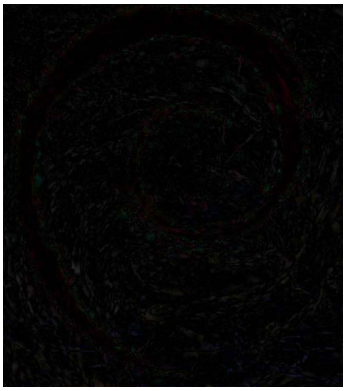

Figure 17: Image de différence après interpolation optimisée

A travers les résultats, on voit bien la différence des deux méthodes sur les bords. A l'aide de l'algorithme optimisé, on obtient une image bien restorée en choisissant uniquement 3% de points dans l'ensemble d'entraînement. Cette méthode est donc aussi utile pour la compression de l'image.
Néanmoins, nous remarquons que l'erreur sur le canal de rouge augmente après l'optimisation. Ceci s'explique du fait que notre objectif est de minimiser l'erreur moyenne sur les trois canaux, donc un compromis entre eux. Il s'applique donc parfois difficilement à une image à forte dominante rouge.

Projet Github

Le lien suivant permet d'accéder au code C++ associé aux différentes méthodes présentées ici:

https://github.com/Tong-ZHAO/sibson_interpolation

References

[1] R. Sibson, *A brief Description of Natural Neighbour Interpolation, Interpreting Multivariate Data* Woley, 1981

[2] G. Farin, *Surfaces over Dirichlet Tessellations*, CAGD, 1990

[3] *CGAL 2D and Surface Function Interpolation*