INRA centre de Jouy en Josas
Unité MIA
Domaine de Vilvert
78350 Jouy-en-Josas

Méthodes d'intégration pour le calcul des flux de gènes entre parcelles dans un paysage agricole

Ying FU

Groupe 9, Deuxième Année
Ecole Centrale de Nantes
Année 2004-2005

Mai - Août 2005

Responsable INRA : **M. Hervé MONOD, M. Kiên KIÊU**
Responsable Ecole Centrale de Nantes : **M. Hervé OUDIN**

Remerciements

Je tiens à remercier toutes les personnes qui m'ont permis d'effectuer mon stage dans de bonnes conditions à commencer par mes maîtres de stage Monsieur Hervé MONOD et Monsieur Kiên KIÊU, pour leur gentillesse, leur disponibilité, leurs explications et leurs conseils avisés.

Ensuite, je remercie également :

- L'ensemble du laboratoire de MIA qui m'a accueilli avec sympathie et a permis mon intégration rapide.
- Monsieur Eric MONVERT et les autres informaticiens du laboratoire, pour leurs aides et leurs conseils.
- Les autres stagiaires et thésards de l'unité pour les moments de détente, et aussi parfois pour leur aide.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Présentation de l'INRA

1.1 Quelques généralités

L'Institut National de la Recherche Agronomique, créé en 1946, est un établissement public à caractère scientifique et technologique, placé sous la tutelle des ministères de la Recherche et de l'Agriculture. Il est chargé de produire et diffuser des conaissances scientifiques et des innovations en tenant compte des exigences de la recherche et des demandes de la société. L'INRA a par ailleurs pour mission de contribuer à l'expertise publique, à la formation et à la promotion de la culture scientifique.

Actuellement, il est formé de :

- 8560 agents dont 1850 scientifiques, 2270 ingénieurs et 4440 techniciens et administratifs.
- 14 départements scientifiques dans le domaines de l'agriculture, de l'alimentation et de l'environnement.
- 21 centres régionaux répartis sur près de 200 sites en France.
- 257 unités de recherche dont 145 unités mixtes de recherche (UMR) associant l'INRA d'autres organismes de recherche ou d'enseignement supérieur et 112 unités propres de recherche.
- 50 unités expérimentales.

Les principaux axes de recherche développés par l'INRA sont :

L'environnement et l'espace rural

L'INRA consacre plus de 20 % de ses moyens à l'étude des écosystèmes cultivés, forestiers et naturels.

L'alimentation humaine et la sécurité des aliments

L'INRA développe des axes de recherche sur la nutrition humaine et les relations entre l'alimentation et la santé.

La bio-informatique

Cette nouvelle discipline doit répondre à l'accroissement extraordinaire de la quantité de données produites par la génomique, par les sciences de l'environnement ou par l'épidémiologie.

La biologie intégrative

L'INRA va renforcer sa capacité de recherche pour la connaissance des génomes, des organismes complets et des populations, tant végétaux qu'animaux.

Les sciences sociales

L'INRA développe des recherches sur la compréhension du monde rural pour fournir aux acteurs privés et publics des informations qui les aident à étayer leurs décisions.

1.2 Le centre de Jouy-en-Josas

Le centre INRA de Jouy-en-Josas abrite 29 laboratoires de recherche scientifique sur les productions animales et l'agroalimentaire : génétique des animaux d'élevage, physiologie, pathologie animale, hydrobiologie, microbiologie, technologie alimentaire, nutrition humaine, sécurité des aliments, biométrie...

Les recherches du centre ont favorisé l'expansion et la qualité des productions animales françaises, puis, à la suite, celles des industries du lait et de la viande. Les travaux scientifiques ont été conduits en collaboration étroite avec les professionnels du monde de l'agriculture et des industries qui y sont liées.

Depuis un demi-siècle, ses études ont contribué à l'amélioration de la productivité des élevages, à la diversification des productions, à la qualité des produits. A titre d'exemples, citons le doublement de la production moyenne de lait par vache laitière, la réduction de moitié de l'adiposité des carcasses de porc, l'émergence de produits nouveaux tels le canard de chair, de nombreux fromages et produits lactés, la maîtrise des techniques de conservation du lait...Ces travaux scientifiques ont par ailleurs servi de base à la création de plusieurs centres INRA en région : Tours pour la physiologie et la pathologie de la reproduction des mammifères d'élevage, et l'aviculture, Clermont-Ferrand pour l'élevage des ruminants et la technologie de la viande, Toulouse pour la génétique des animaux, notamment du mouton, Rennes pour l'élevage du porc et de la vache laitière ainsi que pour la technologie du lait.

L'activité du centre est maintenant caractérisée par une utilisation de plus en plus marquée des techniques de biologie cellulaire et moléculaire, et par la création de modèles animaux originaux. Cette dernière décennie, les recherches en microbiologie alimentaire se sont développées ainsi qu'en nutrition humaine. Le centre est devenu un pôle mondial de biotechnologies autour de nombreux thèmes : développement de l'embryon animal, transfert de gènes, clonage animal, cartes génétiques microbiennes et animales, vaccins, flore du tube digestif humain, ferments lactiques pour l'industrie, ingénierie des protéines. De par ces thématiques-mêmes, un réseau s'est tissé avec le secteur de la santé en matière de reproduction animale, d'immunologie, de nutrition humaine préventive, de sécurité des aliments, d'imagerie médicale. Par exemple, le centre a mis au point des modèles -in vitro ou in vivo sur l'animal- pour étudier les effets de l'alimentation sur l'ostéoporose, la genèse de calculs biliaires ou de cancers, le développement du cerveau, les processus de vieillissement.

1.3 Le département de MIA

Le département de Mathématiques et Informatique Appliquées (MIA) est un département de recherche en mathématique, informatique et intelligence artificielle au sein de l'INRA. Il est présent dans plusieurs centres régionaux.

Les programmes de recherche du département sont orientés par des questions concernant l'environnement, l'agriculture et l'alimentation sur lesquelles le département travaille en collaboration avec les autres départements de recherche de l'INRA (recherches finalisées).

Des recherches originales en mathématique, informatique et intelligence artificielle sont menées dans les unités du département et en collaboration avec des chercheurs de ces disciplines dans les universités et dans d'autres instituts de recherche (recherches disciplinaires).

1.4 Le laboratoire de MIA de Jouy-en-Josas

C'est le laboratoire qui m'a accueilli lors de ce stage. Il est divisé en deux pôles de recherche et une équipe de gestion de l'unité (l'équipe *LogInf*).

Le pôle *MathCell* est spécialisé dans les mathématiques pour la biologie cellulaire. Il comprend une partie d'analyse d'image en biologie et la stéréeologie. l'objectif est de faciliter l'analyse des systèmes complexes pour lesquels les informations se présentent sous forme d'images. L'analyse de gels d'électrophorèse 2D en génomique fonctionnelle est également un thème d'étude de l'équipe. Sur le plan méthodologique, l'équipe vise à résoudre des problèmes d'analyse d'images, apparentés à des problèmes inverses, par des approches probabilistes ou variationnelles. Pour répondre à ces objectifs établis dans le contexte de la biologie cellulaire et moléculaire, l'équipe dispose de compétences variées : analyse d'images (modélisation markovienne, analyse variationnelle, apprentissage, analyse du mouvement et de la déformation, détection et suivi d'objets), stéréologie (précision d'estimateurs stéréologiques, échantillonnage systématique, géométrie stochastique), analyse et modélisation de systèmes nerveux (système olfactif), informatique et développement.

Le pôle *MathRisq* est spécialisé dans les mathématiques par l'évaluation des risques avec pour principales applications les risques alimentaires et épidémiologiques. L'équipe a pour objectifs de collaborer avec les chercheurs de l'INRA ou proches de l'INRA sur l'utilisation de méthodes quantitatives dans les domaines de l'épidémiologie animale et de la sécurité alimentaire, de développer les méthodes mathématicoinformatiques, utiles à court ou moyen terme pour remplir le premier objectif. Les domaines de compétences de ses membres sont principalement : les processus de branchement, l'automatique, la dynamique des populations (EDO,EDP), les réseaux bayésiens, la planification expérimentale, les modèles non-linéaires, la régression PLS. L'équipe a également acquis une expérience dans la gestion de bases de données géoréférencées.

Le pôle de recherche a aussi pour objectif de développer et appliquer des méthodes de modélisation et de statistique pour l'étude de phénomènes agronomiques et écologiques liés à l'environnement. Elle s'intéresse en particulier aux phénomènes de dissémination de gènes d'espèces cultivées, dans le cadre de collaborations pluri-disciplinaires. C'est dans ce cadre là que mon stage s'est déroulé.

Enfin, l'équipe *LogInf* s'occupe de la gestion de l'unité et de la logistique. Elle gère le secrétariat, l'informatique ou encore la bibliothèque de l'unité.

Chapitre 2

Projet de stage - présentation du problème

Certains modèles spatiaux de dynamique des populations intègrent des flux de pollen entre des unités de surface du paysage. L'exemple qui a inspiré cette réflexion est le modèle Genesys de dynamique des populations cultivées et spontanées de colza entre les parcelles d'un paysage et leurs bordures [**?**][**?**].

La dispersion du pollen est modélisée par une fonction ϕ dite fonction de dispersion individuelle. Si d est la distance entre deux points du paysage x et y, $\phi(d)$ est la proportion de pollen émis par une plante située en x et qui arrive au niveau des fleurs femelles d'une plante située au point y. En général pour le colza, la dispersion de pollen est isotrope. C'est pourquoi la fonction de dispersion ne dépend que de la distance séparant les deux points x et y. Dans un cadre plus général la fonction de dispersion prend comme argument un vecteur (bi-dimensionnel) et la proportion de pollen émis par une plante située en x et qui arrive au niveau des fleurs femelles d'une plante située au point y est notée $\phi(y - x)$.

Dans le modèle Genesys, la parcelle est traitée comme une unité de paysage, et l'une des étapes du modèle nécessite le calcul de la proportion de pollen provenant d'une parcelle A et arrivant au niveau des fleurs femelles d'une parcelle B. Cette proportion, que nous noterons $\mathcal{A}$, se calcule en intégrant la fonction de dispersion sur tous les couples de points (x, y) tels que $x \in A$ et $y \in B$. Cette intégrale est de dimension 4 puisque x et y sont des points variant dans un espace bi-dimensionnel. De plus, $\mathcal{A}$ doit être calculée pour l'ensemble des paires de parcelles du paysage. Il s'agit en fait de l'étape la plus lourde du modèle en terme de calcul.

Le rapport [**?**] présente un état des réflexions sur des méthodes de calcul de $\mathcal{A}$ précises et économiques en temps calcul, basées sur des résultats de géométrie algorithmique. Une première étape consiste à réduire la dimension de l'intégrale à calculer. Cette réduction se fait en exploitant les propriétés d'invariance de la fonction de dispersion. Le prix de cette réduction est l'apparition, dans l'intégrale réduite, de quantités géométriques dont le calcul n'est pas trivial. Ce calcul peut être effectué en recourant à des méthodes de géométrie algorithmique. Une seconde étape consiste en une intégration en dimension deux sur un polygone, pour laquelle plusieurs méthodes sont envisageables.

Mon projet de stage a consisté à préciser les étapes de ces calculs et à les programmer en langage C, puis à étudier le comportement de la méthode sur quelques exemples.

Chapitre 3

Algorithmes et programmation

3.1 Hypothèses de travail et données pour les applications

On se place dans le cas général où la fonction de dispersion ϕ n'est pas nécessairement isotrope. On la considère donc comme une fonction définie sur $\mathbb{R}^2$. Deux parcelles A et B sont considérées comme des polygones. La proportion $\mathcal{A}$ de pollen reçue par les plantes femelles de la parcelle B en provenance de la parcelle A s'écrit donc sous la forme

$$\mathcal{A} = \int_A \int_B \phi(y - x) dy dx. \tag{3.1}$$

Pour les parcelles, on utilise un jeu de données fourni avec démonstration d'Ortho, la base de données de l'IGN. On dispose ainsi de 66 parcelles (polygonales) définies via les coordonnées de leurs sommets, voir Fig 3.1.

Pour la fonction de dispersion individuelle ϕ, on utilise une fonction décrite par Etienne Klein [**?**] telle qu'elle est actuellement implémentée dans Genesys [**?**] (Communication personnelle de N. Colbach) :

$$\phi(t) = \begin{cases} d + er + fr^2 & \text{si } 0 \leq r \leq 1,5 \\ \frac{b}{1+r^c/a} & \text{si } r > 1,5 \end{cases} \quad , r = \| t \| . \tag{3.2}$$

avec $a = 3,80$, $b = 0,03985$, $c = 3,12$, $d = 0,340$, $e = -0,405$, $f = 0,128$. Les distances sont exprimées en mètres.

En plus de la fonction de dispersion (3.2), une autre fonction de dispersion très utile même si elle n'est pas du tout réaliste : $\phi \equiv 1$. En effet, pour $\phi \equiv 1$, l'intégrale $\mathcal{A}$ vaut tout simplement $aire(A) \times aire(B)$. Cette fonction de dispersion permet donc de tester les algorithmes.

3.2 Réduction de l'intégrale de quatre à deux dimensions

La proportion de pollen reçu $\mathcal{A}$ se calcule en intégrant $\phi(y - x)$ pour tous les couples $(x, y) \in A \times B$.

Au lieu de parcourir les couples de points en choisissant chaque extrémité indépendamment dans A et B, on peut regrouper les couples de points séparés par le même vecteur. On notera que la fonction de dispersion est constante sur un tel sous-ensemble de couples de points.

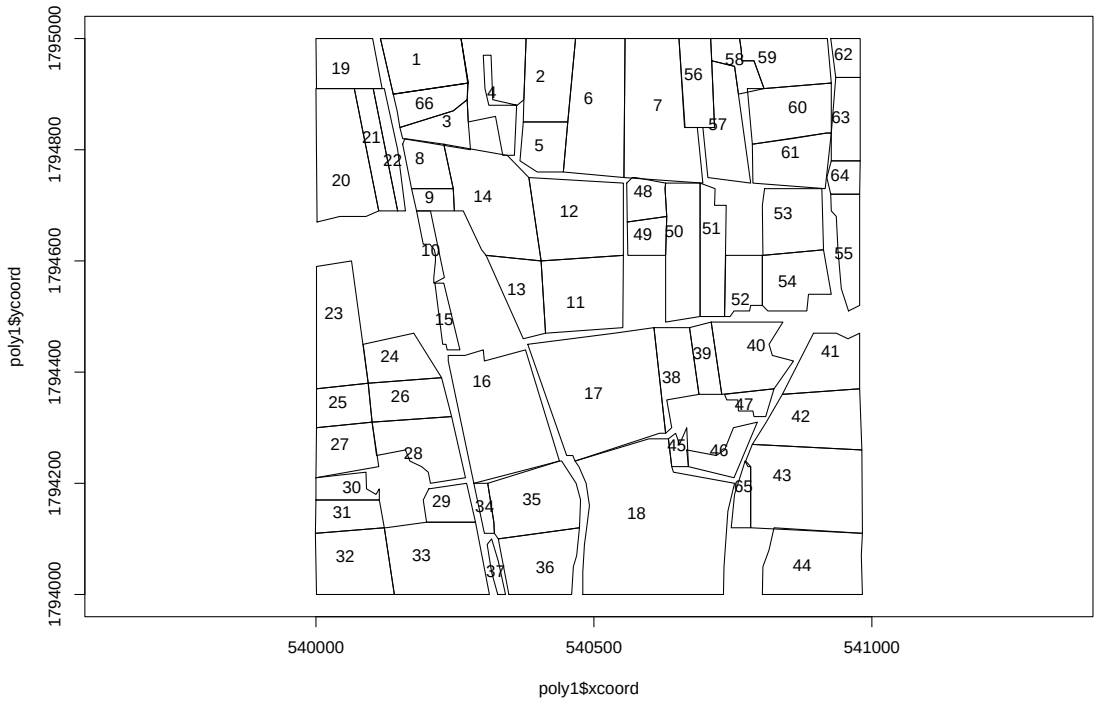


FIG. 3.1 – Échantillon de 66 parcelles. Les coordonnées sont exprimées en mètres.

On effectue le changement de variable $(x, y) \rightarrow (x, t = y - x)$. L'ensemble

$$\{y - x | x \in A, y \in B\}$$

peut se réécrire sous la forme $\check{A} \oplus B$ où $\check{A} = -A$ et $\oplus$ désigne la somme de Minkowski (la somme de Minkowski de deux ensembles est simplement l'ensemble obtenu en sommant les points des deux ensembles). D'autre part, pour tout vecteur $t \in \check{A} \oplus B$, on a

$$x \in A, y = x + t \in B \Leftrightarrow x \in A \cap (B - t),$$

Par ce changement de variable, on réécrit l'intégrale $\mathcal{A}$ définie en équation (3.1) sous la forme

$$\mathcal{A} = \int_{\check{A} \oplus B} \int_{A \cap (B-t)} \phi(t) dx dt \tag{3.3}$$

$$= \int_{\check{A} \oplus B} \underbrace{\int_{A \cap (B-t)} dx}_{aire(A \cap (B-t))} \phi(t) dt \tag{3.4}$$

Alors, on a

$$\mathcal{A} = \int_{\check{A} \oplus B} aire(A \cap (B - t)) \phi(t) dt \tag{3.5}$$

3.3 Méthodes de géométrie algorithmique utilisées

3.3.1 Décomposition d'un polygone nonconvexe en sous-polygones convexes

Cette décomposition est nécessaire car les étapes suivantes sont plus simples si l'on travaille avec des polygones convexes. On a trois étapes :

Triangulation [?]

Voir Algorithme 1.

Algorithm 1 TRIANGULATION

Initialize the ear tip status of each vertex.
while $n > 3$ **do**
  Locate an ear tip v_2.
  Output diagonal $v_1 v_3$.
  Delete v_2.
  Update the ear tip status of v_1 and v_3.
end while

Remarque : Dans l'algorithme, « ear tip »designe un sommet du polygone dont les deux sommets voisins forment une diagonale du polygone.

Hertel-Mehlhorn Algorithm

Supprimer les diagonales nonessentielles. Nous utilisons le programme **HMAlgor()** récupéré sur l'internet[1].

[1] http ://www.bringyou.to/compgeom/triphilvaz.c

Enregistrement des polygones décomposés

Les sous-polygones convexes de la décomposition n'ont pas été enregistrés par les programmes précédents. L'algorithme que j'ai élaboré pour le faire est décrit ci-dessous.

On construit une structure qui s'appelle **DiagonalTest** , avec une variable de ce type : **Diag[MAX_VERTICES]**. Ce type de structure contient une variable booléenne **use**, deux variables entières **from** et **to**, qui indiquent les indices de point de départ et de point d'arrêt d'une diagonale.

Après l'exécution de Triangulate() et HMAlgor(), on obtient **d** (nombre de diagonales qui restent). Alors on enregistre les diagonales dans le tableau **Diag[d]**, et aussi les sommets qui apparaissent dans les diagonales, dans l'ordre décroissant, dans un tableau d'entiers **vdiag[nv]** avec **nv** : nombre de ces sommets.

Voir l'exemple Fig 3.2, Algorithme 2, programme A.5 (page 50).

Algorithm 2 Enregistrer les polygones décomposés

```
Initiate the number of the subpolygons np=0.
Save the subpolygons who have only one diagonal shared with others.
Update the status of the diagonal, and number of the subpolygons.
for all unused diagonal do
  Save the subpolygon which starts with this unused diagonal.
  Update the status of the diagonal, and number of the subpolygons.
end for
```

Comme toutes les fonctions et les actions qui suivent sont dans le cas où le polygone est convexe, on ne considère plus les polygones nonconvexes.

3.3.2 Somme de Minkowski de 2 polygones

Nous utilisons l'algorithme de O'ROURKE (1998, page306). [2]

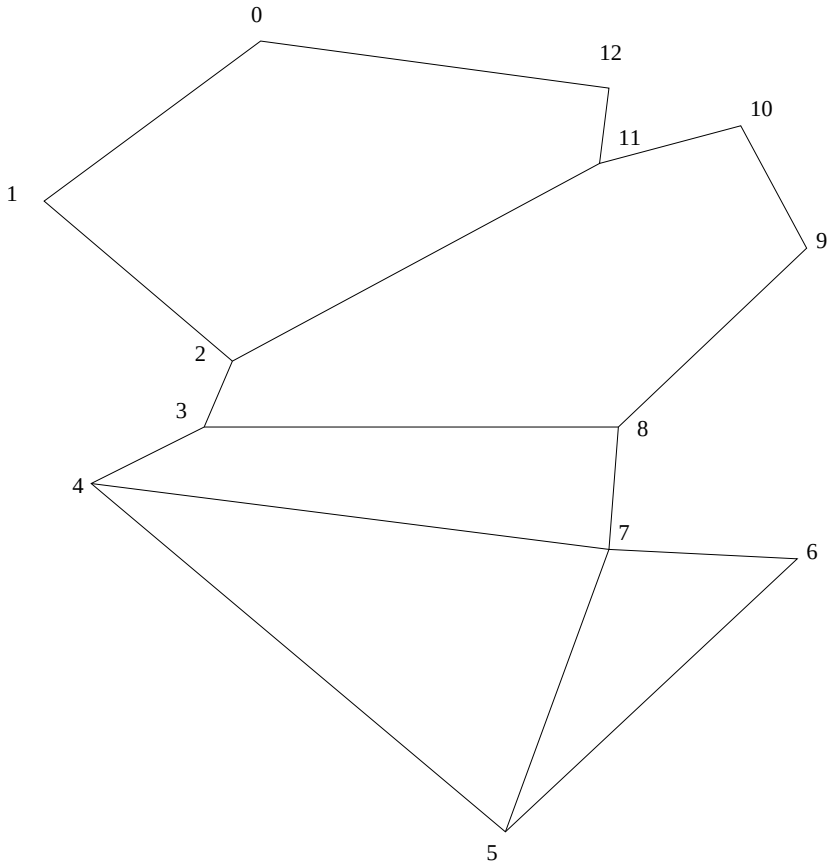
Voir programme A.7 (page 57).

3.3.3 Calcul de l'intersection de 2 polygones

L'intersection de deux polygones A et B est un polygone dont les sommets sont des sommets de A ou B ou des intersections de leurs arêtes.

Nous utilisons l'algorithme de O'ROURKE (1998, page253). [3]

Voir Algorithme 3, programme A.9 (page 63).



Diag[d]:

2 − 11 11 − 2
3 − 8 8 − 3
4 − 7 7 − 4
5 − 7 7 − 5

vdiag[nv]:

11 − 8 − 7 − 5 − 4 − 3 − 2 − 11

FIG. 3.2 – Enregistrement des sous-polygones convexes. D'abord on enregistre les sous-polygones qui ne contiennent qu'une diagonale. Ils sont repérés par le fait que deux nombres consécutifs dans $vdiag[nv]$ forment une diagonale présente dans $Diag[d]$.Dans l'exemple : 2-11-12-0-1 et 7-5-6. Puis, pour chaque diagonale dans $Diag[d]$ non encore utilisée, on enregistre un nouveau sous-polygone. Par exemple on considère la diagonale 11-2 : il n'existe pas de $Diag[k].from = 2$, donc on prend le sommet suivant du polygone, on a 11-2-3. On trouve la diagonale non utilisée 3-8, alors après avoir vérifié quelques conditions on prend le sommet 8, on a 11-2-3-8. Ensuite on trouve de la même façon 9-10-11. Comme $11 = index[0]$, on s'arrête et on a finalement 11-2-3-8-9-10. On peut trouver de la même manière tous les sous-polygones.

Algorithm 3 INTERSECTION OF CONVEX POLYGONS

```
Choose A and B arbitrarily.
repeat
  if A intersects B then
    Check for termination.
    Update an inside flag.
  end if
  Advance either A or B, depending on geometric conditions.
until both A and B cycle their polygons
Handle P ∩ Q = ∅ and P ⊂ Q and P ⊃ Q cases.
```

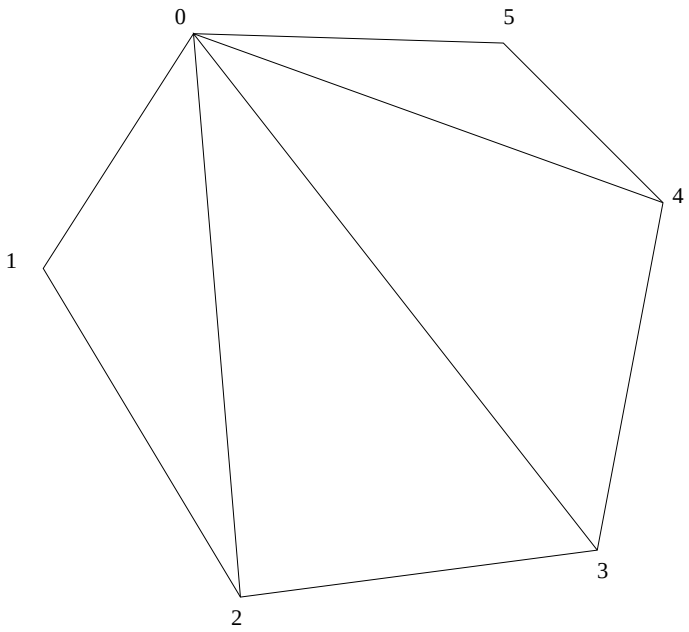


FIG. 3.3 – Calcul de l'aire d'un polygone, par décomposition en triangles

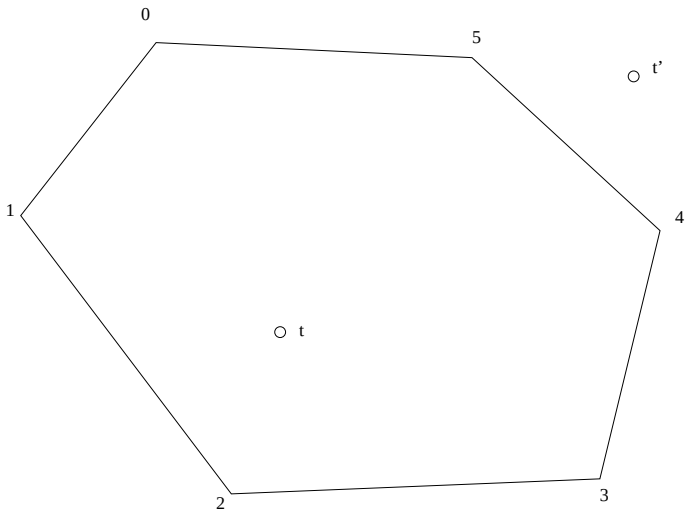


FIG. 3.4 – InPolyConvex

3.3.4 Calcul de l'aire d'un polygone

On divise le polygone en triangles, et on somme les aires des triangles.

Voir Fig 3.3, programme A.10 (page 69).

3.3.5 Tester si un point est à l'intérieur d'un polygone

Nous pouvons voir que (Fig 3.4), le point testé est à l'intérieur d'un polygone convexe si et seulement si ce point est toujours à gauche lorsque l'on parcourt le périmètre du polygone.

Voir programme A.8 (page 62).

3.4 Calcul de l'intégrale à deux dimensions

Deux méthodes d'intégration bidimensionnelle pour le calcul de la Formule 3.5 :

$$\mathcal{A} = \int_{\check{A}\oplus B} aire(A \cap (B - t))\phi(t)\,dt$$

ont été implémentées au cours de mon stage. La première est basée sur une discrétisation par grilles régulières disposées aléatoirement sur la surface à intégrer. C'est une méthode robuste et

[2] « Let A and B be two sets of points in the plane.If we establish a coordinate system, then the points can be viewed as vectors in that coordinate system. Define the *sum* of A and B in the most natural manner possible : $A \oplus B = \{x + y | x \in A, y \in B\}$,where $x + y$ is the vector sum of the two points. This is known as the *Minkowski sum* of A and B. »

[3] « Assume the boundaries of the two polygon P and Q are oriented counterclockwise as usual, and let A and B be directed edges on each. The algorithm has A and B 'chasing' one another, adjusting their speeds so that they meet at every crossing of ∂P and ∂Q. »

« Let a be the index of the head of A, and b the head of B. If B 'aims toward' the line containing A, but does not cross it, then we want to advance B in order to 'close in' on a possible intersection with A. This is the essence of the advance rules. »

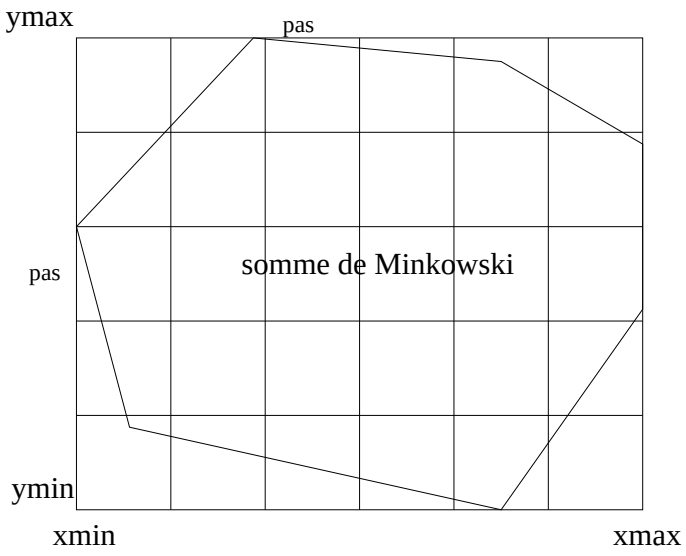


FIG. 3.5 – Boîte de la somme de Minkowski et grille avant tirage aléatoire de sa position

non biaisée, simple à implémenter, mais à priori peu efficace d'un point de vue du temps de calcul. La seconde repose au contraire sur un algorithme rapide [**?**], que l'on a récupéré par internet, l'algorithme Cubpack++.

3.4.1 Intégration par grille

On a deux polygones A et B, avec leurs nombres de sommets a et b. D'abord, on calcule la somme de Minkowski $\check{A} \oplus B$. On en déduit la plus petite boîte rectangulaire de côtés parallèles aux axes qui inclut la somme de Minkowski. Puis on effectue une intégration simple sur ce rectangle avec un coefficient w qui vaut 1 si le point est dans la somme de Minkowski, 0 sinon.

L'intégration proprement dite se fait à partir de grilles régulières de points, de pas commun à fixer par l'utilisateur. Afin d'obtenir un estimateur non biaisé de la valeur à calculer :

- On dispose chaque grille de façon aléatoire sur la surface d'intégration : la position du point d'origine de la grille est déterminée par tirages pseudo-aléatoires uniformes de ses coordonnées dans les intervalles $[0, px]$ et $[0, py]$, où px désigne le pas de la grille en x et py le pas en y.
- On effectue des répétitions, c'est-à-dire que l'on utilisé plusieurs grilles (nombre à fixer par l'utilisateur), disposées indépendamment les unes des autres.
- On estime l'intégrale par la moyenne des résultats obtenus à chaque répétition.

Voir l'exemple Fig 3.5.

3.4.2 Intégration par l'algorithme Cubpack++

Cubpack++ est une méthode d'intégration de fonctions définies sur des triangles quelconques ou sur d'autres formes géométriques. Des détails sont donnés dans la référence[**?**].

Pour l'appliquer à notre problème, on utilise une triangulation de la somme de Minkowski $\check{A} \oplus B$, telle que celle représentée en Fig 3.3.

3.5 Organisation du programme principal

Le programme principal est conçu pour traiter un ensemble de parcelles, avec deux possibilités pour l'utilisateur :

- calculer l'intégrale recherchée pour deux parcelles particulières (non nécessairement distinctes) ;
- calculer l'intégrale recherchée pour toutes les paires de parcelles (y compris les paires de parcelles égales).

Pour éviter de répéter inutilement des calculs, le test de convexité des parcelles et leur décomposition éventuelle en sous-polygones convexes sont effectués en début de programme. Puis on construit une boucle pour toutes les paires de parcelles pour le calcul de l'intégrale. Voilà l'algorithme du programme principal : Algorithme 4.

Algorithm 4 Programme principal

```
Lire toutes les coordonnées des parcelles. Pour chaque parcelle ( polygon ), on teste la convexité,
si le polygon est nonconvexe, on le décompose, et les enregistre dans Poly[npoly].
if Choix du calcul de l'intégrale de deux parcelles then
  Donner les indices de deux parcelles
  Calcul de l'intégrale, obtenir les sorties
else
  {Choix du calcul de l'intégrale de toutes les paires de parcelles}
  for c = 0 to npoly do
    for d = 0 to npoly do
      for nr = 0 to nb de répétition do
        {Pour la méthode de l'intégrale par grilles}
        Calcul de l'intégrale.
      end for
      Obtenir les sorties
    end for
  end for
end if
```

Voir programme A.1 (page 37).

Chapitre 4

Application numérique et analyse des résultats

4.1 Objectifs et cas considérés

Dans ce chapitre, nous étudions le comportement du programme et des deux algorithmes d'intégration, sur des paires des parcelles extraites des 66 parcelles de la Fig 3.1 et avec la fonction ϕ donnée en Section 3.1.

On considère 4 cas : 1) les deux parcelles sont égales et convexes ; 2) les deux parcelles sont égales et nonconvexes ; 3) les deux parcelles sont voisines et toutes les deux sont convexes ; 4) les deux parcelles sont voisines et une d'entre elles est nonconvexe. De plus, on considère un cas particulièrement complexe lorsque les deux parcelles sont toutes les deux la parcelle de forme très irrégulière numéro 4.

4.2 Influence du pas d'intégration pour la méthode par grille

On étudie l'effet du pas d'intégration sur la précision de l'algorithme et sur le temps d'exécution : On choisit 10 pas différents pour la grille d'intégration variant de 0.25m à 1.51m. On fixe le nombre de répétitions de l'algorithme à 20. On s'intéresse aux sorties suivantes : l'estimateur de la moyenne $\hat{\mu}$; l'estimateur de l'écart-type $\hat{\sigma}$ entre répétitions ; le coefficient de variation $\hat{\sigma}/\hat{\mu}$; l'intervalle de confiance $[\hat{\mu}_{inf}, \hat{\mu}_{sup}]$. (Tab 4.1)

Le coefficient de variation de l'estimateur basé sur 20 répétitions, égal à $\frac{\hat{\sigma}/\sqrt{20}}{\hat{\mu}}$, est représenté dans les Figures 4.1 à 4.5.

4.3 Comparaison entre l'intégration par grille et l'intégration par Cubpack++

Pour améliorer le programme, on a considéré d'autres méthodes d'intégration, et on a choisi la méthode Cubpack++, qui est écrite en C++. Et ici, on compare les intervalles de confiance, avec ceux obtenus en intégrant par gille.

On trouve ci-dessous les tableaux (Tab 4.2 - Tab 4.5) et les figures (Fig 4.2 - Fig 4.5) de résultats pour les cas précédents, avec la précision de 0.01 à 0.001 pour chaque cas.

Paire de parcelles	**Pas(m)**									
	0.25	0.39	0.53	0.67	0.81	0.95	1.09	1.23	1.37	1.51
cas 1 : 1+1	12368 5.139 0.04155% **12368± 2.919**	12371 22.05 0.1783%	12377 51.41 0.4154%	12363 84.01 0.6796%	12382 172.8 1.395%	12331 150.8 1.223%	12330 231.7 1.879%	12249 307.0 2.506%	12305 571.8 4.647%	12116 1031 8.511%
cas 2 : 14+14	23603 76.19 0.3228% **23603± 43.27**	23615 237.1 1.004%	23545 64.34 0.2733%	23526 105.5 0.4484%	23655 576.3 2.436%	23637 495.2 2.095%	23822 749.8 3.147%	23268 308.8 1.327%	23725 1538 6.482%	23285 1746 7.500%
cas 3 : 11+12	150.05 0.07107 0.04736% **150.05± 0.04**	149.99 0.2486 0.1657%	150.00 0.5753 0.3836%	150.05 1.096 0.7306%	150.22 1.496 0.9957%	149.81 2.311 1.543%	150.78 3.854 2.556%	151.17 4.637 3.067%	151.49 7.074 4.670%	149.34 8.127 5.442%
cas 4 : 56+57	192.02 191.5 99.75% **192.02± 108.76**	139.59 32.25 23.11%	130.45 7.163 5.461%	157.27 96.81 61.55%	140.77 39.27 27.98%	128.12 1.908 1.489%	127.47 3.292 2.583%	133.67 22.07 16.51%	127.99 4.374 3.417%	129.57 7.688 5.934%
cas 5 : 4+4 cas particulier	15510 61.81 0.3985% **15510± 35.11**	15597 294.5 1.600%	15524 190.0 1.224%	15543 442.5 2.847%	15469 138.5 0.8955%	15454 171.3 1.108%	15480 283.1 1.829%	15356 397.2 2.586%	15359 254.9 1.659%	15488 600.7 3.879%

TAB. 4.1 – Tableau des résultats dans différents cas. Pour chaque cas, la première ligne présente le résultat moyen ; la deuxième ligne présente l'écart-type ; la troisème ligne présente le coefficient de variation ; la quatrième ligne (en gras) présente l'intervalle de confiance, mais seulement pour le pas 0.25m. Nombre de répétitions : 20.

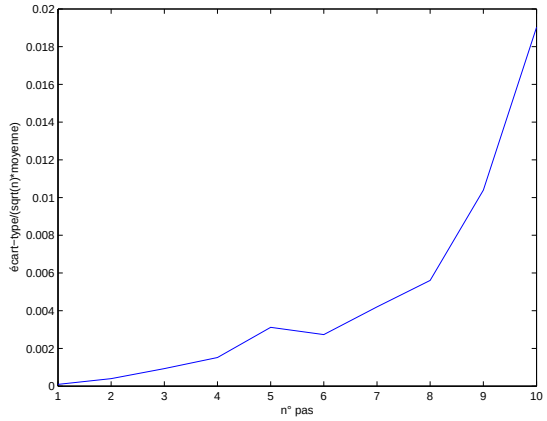


FIG. 4.1 – Influence du pas, cas 1 : deux parcelles égales et convexes

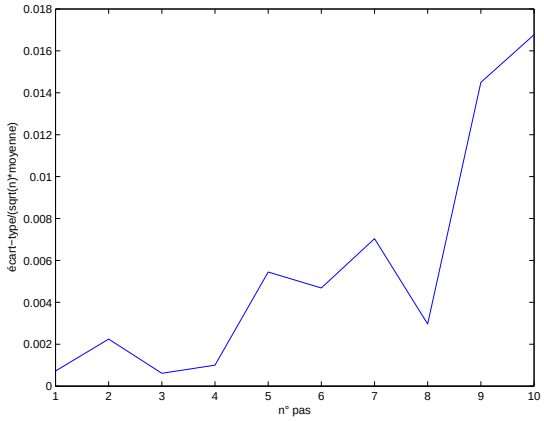


FIG. 4.2 – Influence du pas, cas 2 : deux parcelles égales et nonconvexes

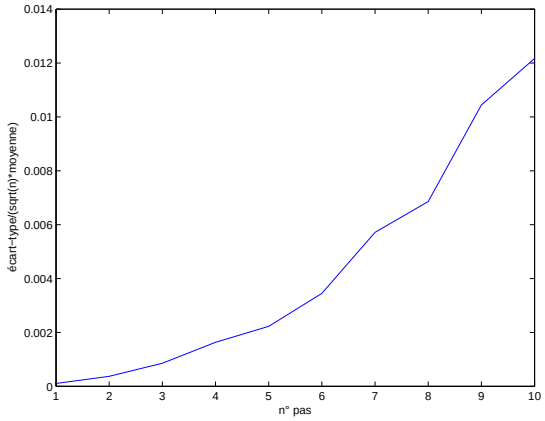


FIG. 4.3 – Influence du pas, cas 3 : deux parcelles voisines et convexes

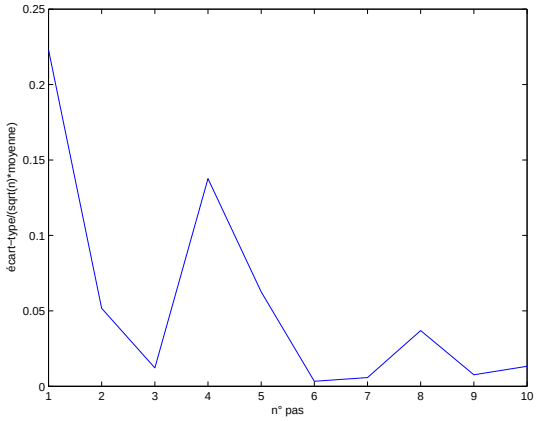


FIG. 4.4 – Influence du pas, cas 4 : deux parcelles voisines et une est nonconvexe

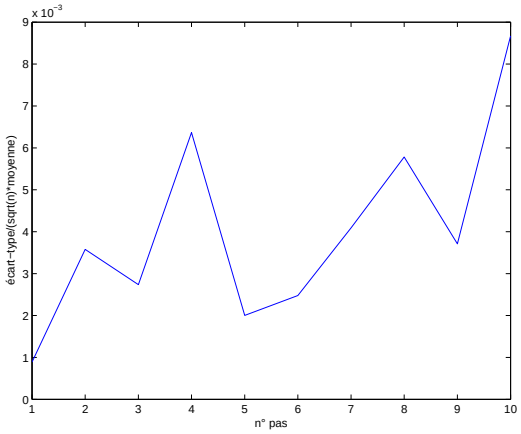


FIG. 4.5 – Influence du pas, cas 5 : deux parcelles égales et nonconvexes et très complexes(4+4)

1+1	Précision	Résultat Moyen	Inervalle de Confiance
1	0.01	12363	[12239.5 12486.8]
2	0.009	12360	[12249.0 12471.4]
3	0.008	12361	[12261.9 12459.6]
4	0.007	12361	[12274.8 12447.9]
5	0.006	12362	[12287.8 12436.2]
6	0.005	12364	[12302.2 12425.8]
7	0.004	12365	[12315.6 12414.5]
8	0.003	12366	[12329.1 12403.3]
9	0.002	12367	[12342.4 12391.9]
10	0.001	12367	[12354.9 12379.6]

TAB. 4.2 – Cubpack_Cas 1 : 1+1 : résultats pour différentes précisions demandées

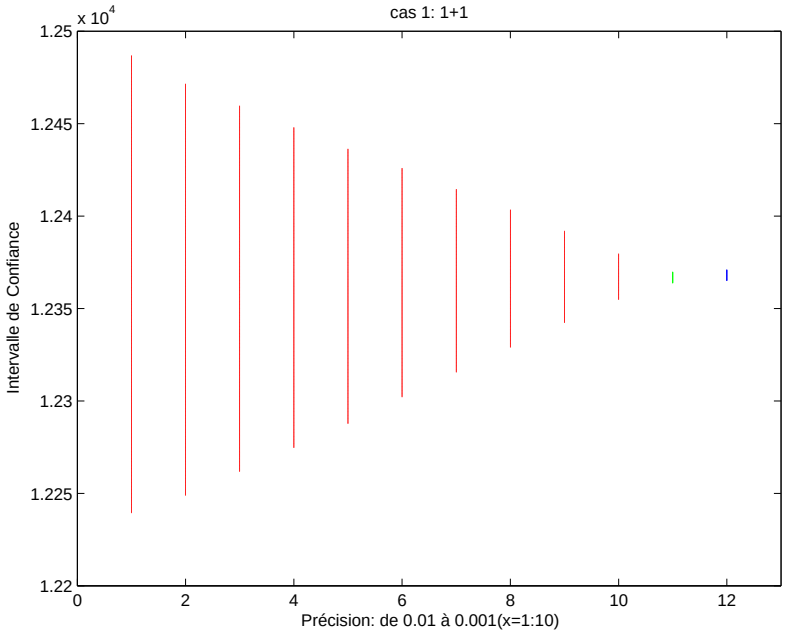


FIG. 4.6 – Cas 1 : 1+1. x=1 :10 : précision de 0.01 à 0.001 ; x=12 : l'intervalle de confiance du calcul sans Cubpack, avec pas=0.25m et 20 répétitions ; x=11 : l'intervalle de confiance du calcul avec Cubpack et avec la même précision que le cas où x=12.

De plus, nous pouvons mesurer le temps de calcul pour chaque cas, avec la même précision que le calcul par grille avec pas = 0.25m.(Tab 4.6)

14+14	Précision	Résultat Moyen	Inervalle de Confiance
1	0.01	23542	[23307.0 23777.9]
2	0.009	23555	[23343.4 23767.3]
3	0.008	23555	[23366.8 23743.7]
4	0.007	23552	[23387.2 23716.9]
5	0.006	23547	[23406.0 23688.6]
6	0.005	23548	[23430.3 23665.7]
7	0.004	23543	[23448.7 23637.0]
8	0.003	23549	[23478.6 23619.9]
9	0.002	23549	[23502.1 23596.3]
10	0.001	23546	[23522.1 23569.2]

TAB. 4.3 – Cubpack_Cas 2 : 14+14 : résultats pour différentes précisions demandées

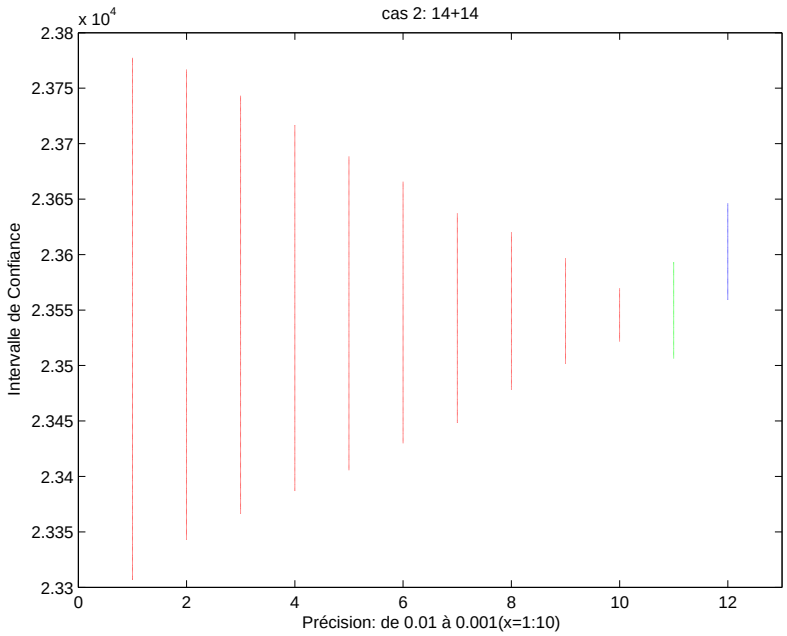


FIG. 4.7 – Cas 2 : 14+14. x=1 :10 : précision de 0.01 à 0.001 ; x=12 : l'intervalle de confiance du calcul sans Cubpack, avec pas=0.25m et 20 répétitions ; x=11 : l'intervalle de confiance du calcul avec Cubpack et avec la même précision que le cas où x=12.

11+12	Précision	Résultat Moyen	Inervalle de Confiance
1	0.01	150.65	[149.140 152.153]
2	0.009	149.20	[147.861 150.546]
3	0.008	149.21	[148.014 150.401]
4	0.007	149.21	[148.163 150.252]
5	0.006	149.61	[148.713 150.509]
6	0.005	150.04	[149.292 150.792]
7	0.004	150.04	[149.442 150.642]
8	0.003	150.04	[149.592 150.493]
9	0.002	150.04	[149.743 150.344]
10	0.001	150.05	[149.897 150.197]

TAB. 4.4 – Cubpack_Cas 3 : 11+12 : résultats pour différentes précisions demandées

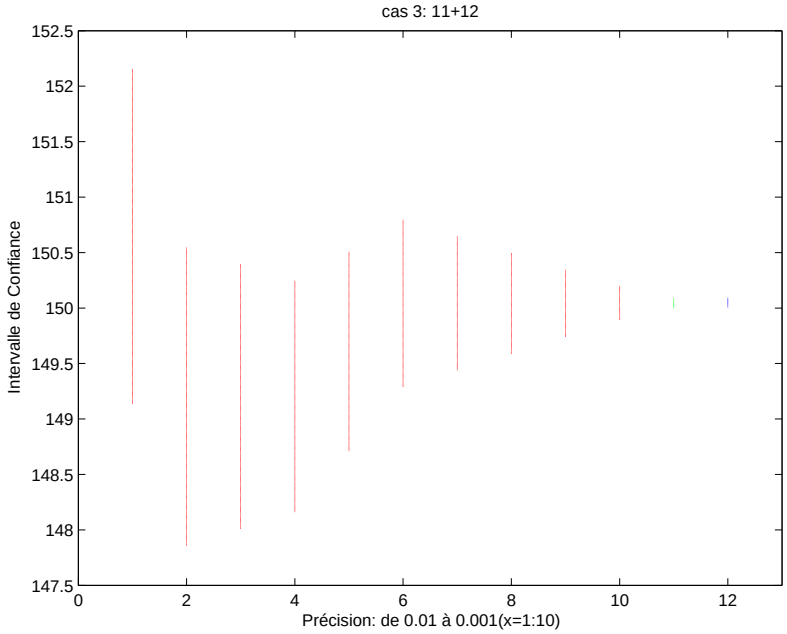


FIG. 4.8 – Cas 3 : 11+12. x=1 :10 : précision de 0.01 à 0.001 ; x=12 : l'intervalle de confiance du calcul sans Cubpack, avec pas=0.25m et 20 répétitions ; x=11 : l'intervalle de confiance du calcul avec Cubpack et avec la même précision que le cas où x=12.

56+57	Précision	Résultat Moyen	Inervalle de Confiance
1	0.01	127.52	[126.244 128.764]
2	0.009	127.57	[126.424 128.720]
3	0.008	127.64	[126.621 128.664]
4	0.007	127.64	[126.749 128.536]
5	0.006	127.64	[126.877 128.409]
6	0.005	127.64	[127.003 128.279]
7	0.004	128.01	[127.496 128.521]
8	0.003	128.02	[127.632 128.400]
9	0.002	128.30	[128.046 128.559]
10	0.001	128.30	[128.168 128.424]

TAB. 4.5 – Cubpack_Cas 4 : 56+57 : résultats pour différentes précisions demandées

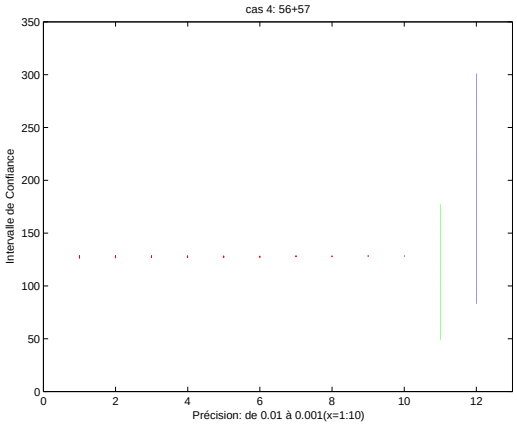


FIG. 4.9 – Cas 4 : 56+57. x=1 :10 : précision de 0.01 à 0.001 ; x=12 : l'intervalle de confiance du calcul sans Cubpack, avec pas=0.25m et 20 répétitions ; x=11 : l'intervalle de confiance du calcul avec Cubpack et avec la même précision que le cas où x=12.

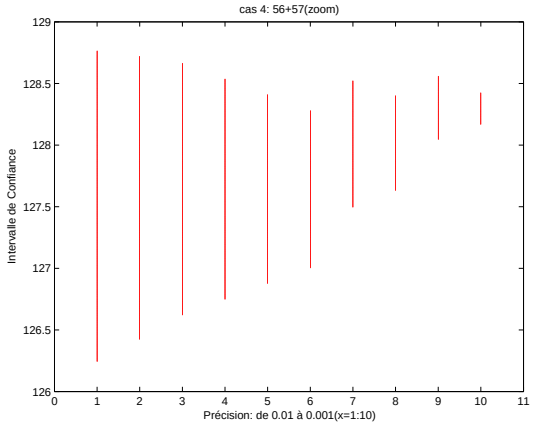


FIG. 4.10 – Cas 4 : 56+57 (zoom). x=1 :10 : précision de 0.01 à 0.001.

Paire de parcelles	**Précision** (%)	**Cubpack++**			**Grille, Pas=0.25m, 20 répétition**		
		Résultat Moyen	Intervalle de Confiance	Temps Calcul(ms)	Résultat Moyen	Intervalle de Confiance	Temps Calcul(ms)
1+1	0.02360	12367	[12363.8 12369.7]	1690	12368	[12365.1 12370.9]	83290
14+14	0.18333	23550	[23506.8 23593.2]	16400	23603	[23559.7 23646.3]	1084230
11+12	0.02705	150.04	[150.003 150.084]	220	150.05	[149.646 150.454]	296390
56+57	56.65	113.37	[49.1458 177.594]	20	192.02	[83.2554 300.785]	248390

TAB. 4.6 – Comparer les deux méthodes d'intégration : avec grille et avec Cubpack++, avec la même précision.

Chapitre 5

Conclusion

Le stage a permis à partir de méthodes de géométrie algorithmique, de mettre au point un programme C pour le calcul de la quantité

$$\mathcal{A} = \int_A \int_B \phi(y - x) dy dx$$

, où A et B sont deux polygones et ϕ une fonction définie sur $\mathbb{R}^2$. Ce programme permettra en particulier de calculer les flux de pollen entre paires de parcelles de formes polygonales quelconques dans un paysage agricole.

Deux méthodes d'intégration ont été implémentées. Les comparaisons numériques, effectuées sur différents types de paires de parcelles, montrent la cohérence des résultats obtenus avec les deux méthodes, ainsi qu'un très net avantage de la méthode Cubpack++ en terme d'efficacité pour le temps de calcul.

Annexe A

Liste des programmes

Les programmes sont répartis dans 5 fichiers : intflux.c, geom.c, zoneintegration.c, intersection.c intflux.h.

A.1 Programme principal

(Dans le fichier “intflux.c”)

```
#--------------------------------------------------------------------#
# D'abord, pour le programme principal, il faut lire tout les       #
#  polygones, tester la convexité, et calculer l'intégration. Aussi #
#  on peut obtenir le temps de calcul.                               #
#--------------------------------------------------------------------#

#include <sys/times.h>
#include <sys/types.h>
#ifdef __ultrix
#define _POSIX_SOURCE
#endif // __ultrix
#include <time.h>
#ifdef __ultrix
#undef _POSIX_SOURCE
#endif // __ultrix

long times_(void)
{
  long t=0;
  struct tms tms;
  times(&tms);

  t = tms.tms_utime + tms.tms_stime;   // this is in  clock_ticks
  // conversion of clock_ticks to milliseconds
  t = (long)((double)(t)*(double)(1000)/(double)(100));
  return t;
}
```

```
int main ()
{
  FILE *fp;
  char str1[80], str2[80], str3[80];
  tPolygoni **Poly;
  tVertex vertices;
  tVertex p;
  POLYGON_STRUCT PolygonVertices[MAX_VERTICES];
  DIAGONAL_STRUCT PolygonDiagonals[MAX_VERTICES];
  int nvertices, ndiagonals;
  int i, j;
  int c, d, nr,k;
  int npoly;
  int est;
  int cas;
  double h, l;              // pas pour l'intégration
  char change;
  Bool convex;
  long Time, OldTime;       // Pour calculer le temps d'exécution
  unsigned long second;

  fp = fopen("poly_coord.txt", "r");

  fseek(fp, -29, SEEK_END);
  fscanf(fp,  "%d",&npoly);
  printf("npoly = %d\n", npoly);
  rewind(fp);
  fscanf(fp, "%s %s %s", str1, str2,str3);

  Poly = (tPolygoni**)calloc(npoly, sizeof(tPolygoni*));
  NEW(vertices, tsVertex);
  vertices->next = vertices->prev = vertices;

  int a[npoly];     // nombre de sous-polygones convexes
  int ni[npoly][MAX_VERTICES];
  double result[npoly*npoly][20];
  double r[npoly*npoly];
  double e[npoly*npoly];
  double area[npoly];

  for(i=0;i<npoly;i++)
    {
      nvertices = ReadVertices(i+1, PolygonVertices,vertices, fp);
      // enregistrer les sommets de l'(i+1)ème polygone dans une liste de chaîne,
      // qui commence par le pointeur: vertices.
      p = vertices->next;
      do {
        area[i] = area[i] + Area2i(vertices->v, p->v, p->next->v );
        p = p->next;
      } while (p->next != vertices);
```

```
    convex = Convexity(vertices);
    // si le polygone n'est pas convexe, décomposons-lui.

    if ( !convex )
      {
        printf("%d th polygon is nonconvex\n", i+1);
        ndiagonals = Triangulate(PolygonDiagonals,vertices, nvertices);
        Poly[i] = (tPolygoni*) calloc(30,sizeof(tPolygoni));
        a[i] =  HMAlgor(Poly[i],ni[i],PolygonDiagonals,
                        PolygonVertices,nvertices,ndiagonals);
      }
    else
      {
        // printf("without closed point\n");
        Poly[i] = (tPolygoni*) malloc(sizeof(tPolygoni));
        for (j=0;j<nvertices;j++)
          {
            Poly[i][0][j][X] = PolygonVertices[j].xv;
            Poly[i][0][j][Y] = PolygonVertices[j].yv;
          }
        ni[i][0] = nvertices;
        a[i] = 1;
        printf("%d th polygon is convex\n", i+1);
        // si le polygone est convexe, nous l'enregistrons dans un tableau de
        // polygone avec seulement 1 polygone.
      }
    free(vertices);
    printf("****************************************\n");
  }
fclose(fp);
k=0;
printf(" Choose the one you want:\n
       1. you want to calcule the integral of two parcels;\n
       2. you want to calcule the integrals of all the pairs of parcels.\n");
scanf("%d",&cas);
switch(cas)
{
      case 1:
           printf("Case 1:\n");
           printf(" the first polygone: ");
           scanf("%d", &c);
           printf(" the second polygone: ");
           scanf("%d", &d);
           c=c-1;
           d=d-1;
           printf(" step for integration x axis: ");
           scanf ("%lf", &l);
           printf(" step for intergration y axis: ");
           scanf ("%lf", &h);
```

```
        printf(" how many estimations do you want?");
        scanf("%d",&est);

        second = 0;
        Time = OldTime = times_();

        r[k] = 0;
        for(nr=0;nr<est;nr++)
          {
            result[k][nr] = 0;
             for (i=0;i<a[c];i++)
               for (j=0;j<a[d];j++)
               result[k][nr] = result[k][nr]
               + Integration( Poly[c][i], Poly[d][j], ni[c][i], ni[d][j],h,l);
            printf(" result[%d][%d] = % 8.5g \n",k+1, nr+1, result[k][nr]);
            r[k] = r[k]+result[k][nr];
          }
        r[k]=r[k]/nr;
        Time = times_();

        printf(" result moyenne for the polygones n° %d, %d = %8.5g \n", c+1, d+1,r[k]);
        printf(" area * area = %g\n", area[c]*area[d]/4);
        e[k]=0;
        for(i=0;i<est;i++)
        e[k] = e[k]+pow(result[k][i]-r[k],2);
        e[k] = sqrt(e[k]/(est-1));
        printf("standard deviation = %g\n",e[k]);
        printf("coefficient of variation = %g\n", e[k]/r[k]);
        printf(" Elapsed Time: %lu milliseconds\n", Time-OldTime);
        break;
    default:
        printf("Case 2:\n");
        printf(" how many estimations do you want?");
        scanf("%d",&est);
        h=l=0.25;
        printf(" default step for integration x and y axes: 0.25m.\n");
        printf(" Do you want to change the step ? (y/n):");
        scanf("%s",&change);
        if(change=='y')
          {
            printf(" the new step for x axis:");
            scanf("%lf",&l);
            printf(" the new step for y axis:");
            scanf("%lf",&h);
          }
        for(c=0;c<npoly;c++)
           for(d=c;d<npoly;d++)  // calcul de tout les paires de parcelles
             {
                r[k] = 0;
                for(nr=0;nr<est;nr++)
```

```
                        {
                           result[k][nr] = 0;
                           for (i=0;i<a[c];i++)
                              for (j=0;j<a[d];j++)
                              result[k][nr] = result[k][nr]
                              + Integration( Poly[c][i], Poly[d][j], ni[c][i], ni[d][j],h,l);
                           printf(" result[%d][%d] = %8.5g\n",k+1, nr+1, result[k][nr]);
                           r[k] = r[k]+result[k][nr];
                         }
                      r[k]=r[k]/nr;
                      printf(" result moyenne for the polygones n° %d, %d
                                = %8.5g\n", c+1,d+1,r[k]);
                      printf(" area * area = %g\n", area[c]*area[d]/4);
                      e[k]=0;
                      for(i=0;i<est;i++)
                      e[k] = e[k]+pow(result[k][i]-r[k],2);
                      e[k] = sqrt(e[k]/(est-1));
                      printf("standard deviation = %g\n",e[k]);
                      printf("coefficient of variation = %g\n", e[k]/r[k]);
                      k++;
                   }
   }
  return 0;
}
```

A.2 Fonction de dispersion

(Dans le fichier “intflux.c”)

```
#--------------------------------------------------------------------#
#                 La fonction de dispersion :                        #
#--------------------------------------------------------------------#


double function ( tPointd t )
{
  double f;
  double r;

  r = sqrt(pow(t[X],2.0)+pow(t[Y],2.0));
  if ( r<=1.5 )
    f = 0.340-0.405*r+0.128*pow(r,2.0);
  else
    f = 0.03985/(1+pow(r,3.12)/3.80);

  return f;
}
```

A.3 ReadVertices

(Dans le fichier “geom.c”)

```
#-------------------------------------------------------------------#
# Pour lire les coordonnées de polygone d'un fichier, j'écris la  #
#  fonction ReadVertices(int, POLYGON_STRUCT *, tVertex, FILE *), #
# c-à-d qu'on veut tester sur l'ième polygone dans le fichier.    #
#    Le polygone est enregistré comme une chaîne de sommets,      #
#               avec le premier pointeur vertices.                #
#-------------------------------------------------------------------#

int ReadVertices( int i, POLYGON_STRUCT *PolygonVertices,
                  tVertex vertices, FILE *fp )
{
  tVertex  v;
  int j, ID;
  float x, y;
  int vnum = 0;
  int ixmax, ixmin, iymax, iymin;
  Bool Find;
  Find = FALSE;

 boucle:
  while( fscanf(fp,  "%d %f %f",&ID, &x, &y) != EOF ) {
     // chercher le polygone avec l'ID qu'on espère
     if(ID == i)
       Find = TRUE;
     else
       {
         if(!Find)
           goto boucle; // continuer la recherche
         else goto end; // fin du polygone
       }
     if (Find)
       {
         // enregistrer les sommets
 v = MakeNullVertex( vertices );
 v->v[X] = (int)(x);
 v->v[Y] = (int)(y);

 //if ( ( fabs(x) > SAFE ) || ( fabs(y) > SAFE ) )
 //printf("Coordinate of vertex below might be too large:
                     run with -c flag\n");
       }
   };

 end:
     if (i==1)
       {
```

```
        vertices->next = vertices->next->next;
        vertices->next->prev = vertices;
      }
    //   printf(" after the deletion of the last point:\n");
    // comme dans le fichier, la liste de sommets est fermée, on
    // supprime la dernière ligne de données de ce polygone qui
    // est le même que la première.
    vertices = vertices->next;
    v = vertices;
    do{
      // PolygonVertices[ ]: global variable
      PolygonVertices[vnum].xv = v->v[X];
      PolygonVertices[vnum].yv = v->v[Y];
      v->vnum = vnum++;
      printf("%d %d %d\n",v->vnum, v->v[X], v->v[Y]);
      v = v->next;
    }while(v!=vertices);

   if ( vnum<3 )
     {
       printf("Error in ReadVertices: number of vertices = %d < 3\n",nvertices);
       exit(EXIT_FAILURE);
     }


   i = 0;
   ixmax = ixmin = PolygonVertices[i].xv;
   iymax = iymin = PolygonVertices[i].yv;

  do {
    i = i+1;
    if ( PolygonVertices[i].xv > ixmax)  ixmax = PolygonVertices[i].xv;
    else if ( PolygonVertices[i].xv < ixmin)  ixmin = PolygonVertices[i].xv;
    if ( PolygonVertices[i].yv > iymax)  iymax = PolygonVertices[i].yv;
    else if ( PolygonVertices[i].yv < iymin)  iymin = PolygonVertices[i].yv;
  } while(i<vnum-1);

  printf("%%Bounding box:\n");
  printf("xmax = %d; xmin = %d; difference: %d\n", ixmax, ixmin, ixmax-ixmin);
  printf("ymax = %d; ymin = %d; difference: %d\n", iymax, iymin, iymax-iymin);
  return vnum;
}
```

A.4 Tester convexité

(Dans le fichier “geom.c”)

```
#--------------------------------------------------------------------#
# Après la lecture des sommets de polygone, je teste sa convexité, #
# car les autres fonctions ou actions suivantes sont dans le cadre #
# de polygone convexe. Si ce polygone est nonconvexe, on doit le   #
#              décomposer en polygones convexes.                     #
#                                                                    #
# L'idée de tester la convextité: dans le sens inversé des          #
# aiguilles d'une montre, si tous les points sont à droite du       #
# vecteur de son point précédent à son point suivant, alors, le     #
# polygone est convexe. Si il existe au moins un  point qui est à   #
# gauche de son vecteur, il est nonconvexe.                          #
#--------------------------------------------------------------------#

Bool Convexity ( tVertex vertices )
{
  tVertex v0, v1, v2;
  v1  = vertices;
  do {
    v2 = v1->next;
    v0 = v1->prev;
    if ( LeftOn(v0->v, v2->v, v1->v) )
      return FALSE;
    v1 = v1->next;
  } while (v1!=vertices);
  return TRUE;
}
```

A.5 Décomposer polygone

(Dans le fichier “geom.c”)

```
#-------------------------------------------------------------------#
# Si le polygone est nonconvexe, on doit le décomposer.             #
# Premièrement, je fais la triangulation.                           #
#-------------------------------------------------------------------#

void Triangulate(DIAGONAL_STRUCT *PolygonDiagonals, tVertex vertices,
                 int nvertices )
{
   tVertex v0, v1, v2, v3, v4;   // five consecutive vertices
   tVertex a1, a0;               // two more needed to compute if convex/reflex
   int ndiagonals = 0;
   int n = nvertices;            // number of vertices; shrinks to 3
   Bool earfound;                // for debugging and error detection only
   int i;

   // clear diagonal array
   for (i = 0; i < MAX_VERTICES; i++)
     PolygonDiagonals[i].exist = FALSE;

   EarInit( vertices );

   while (n > 3)
   {
      // Inner loop searches for an ear
      v2 = vertices;
      earfound = FALSE;

      do
      {
         if (v2->ear)
         {
            earfound = TRUE;
            // Ear found. Fill variables
            v3 = v2->next; v4 = v3->next;
            v1 = v2->prev; v0 = v1->prev;

            // (v1,v3) is a diagonal

            ////////////////////////////////////////////////////////////
            //
            // CHANGES TO TRIANGULATE FOR HERTEL-MEHLHORN IMPLEMENTATION
            //
            ////////////////////////////////////////////////////////////

            // save diagonal, check if endpoints convex for later removal
            PolygonDiagonals[ndiagonals].exist = TRUE;
```

```
            // save from endpoint and check if convex
            PolygonDiagonals[ndiagonals].vfrom = v1->vnum;
            a1 = v1->next; a0 = v1->prev;
            PolygonDiagonals[ndiagonals].convexfrom = LeftOn(v1->v,a1->v,a0->v);

            // save to endpoint and check if convex
            PolygonDiagonals[ndiagonals].vto = v3->vnum;
            a1 = v3->next; a0 = v3->prev;
            PolygonDiagonals[ndiagonals].convexto = LeftOn(v3->v,a1->v,a0->v);

            ndiagonals++;  // increment number of diagonals

            ////////////////////////////////////////////////////////
            //
            // END OF CHANGES
            //
            ////////////////////////////////////////////////////////

            // Update earity of diagonal endpoints
            v1->ear = Diagonal( v0, v3, vertices );
            v3->ear = Diagonal( v1, v4, vertices );

            // Cut off the ear v2
            v1->next = v3;
            v3->prev = v1;
            vertices = v3; // In case the head was v2

            n--;

            break;   // out of inner loop; resume outer loop

         } // end if ear found

         v2 = v2->next;

      } while (v2 != vertices);

      if (!earfound)
      {
         printf("%%Error in Triangulate:  No ear found.\n");
         printf("showpage\n%%%%EOF\n");
         QuitProgram(EXIT_FAILURE);
      }

   } // end outer while loop
   return ndiagonals;
}
```

```
#-------------------------------------------------------------------#
# Initialize the ear up status of each vertex                       #
#-------------------------------------------------------------------#


void EarInit ( tVertex vertices )
{
  tVertex v0, v1, v2;

  v1 =vertices;
  do {
    v2 = v1->next;
    v0 = v1->prev;
    v1->ear = Diagonal( v0, v2, vertices );
    v1 = v1->next;
  } while (v1!=vertices);
}
*********************************************************************

Bool Diagonal(tVertex a, tVertex b, tVertex vertices)
{
   return InCone(a, b) && InCone(b, a) && Diagonalie(a, b, vertices);
}
*********************************************************************

Bool InCone(tVertex a, tVertex b)
{
   tVertex a0,a1; // a0,a,a1 are consecutive vertices

   a1 = a->next;
   a0 = a->prev;

   // If a is a convex vertex ...
   if (LeftOn(a->v, a1->v, a0->v))
       return Left(a->v, b->v, a0->v) && Left(b->v, a->v, a1->v);

   // Else a is reflex:
   return !(LeftOn(a->v, b->v, a1->v) && LeftOn(b->v, a->v, a0->v));
}
*********************************************************************

Bool Diagonalie(tVertex a, tVertex b, tVertex vertices)
{
   tVertex c, c1;

   // For each edge (c,c1) of P
   c = vertices;

   do
   {
```

```
      c1 = c->next;
      // Skip edges incident to a or b
      if ((c != a) && (c1 != a) && (c != b) && (c1 != b)
         && Intersect( a->v, b->v, c->v, c1->v ))
         return FALSE;

      c = c->next;

   } while ( c != vertices );

   return TRUE;
}
*****************************************************************

Bool Intersect( tPointi a, tPointi b, tPointi c, tPointi d)
{
  Bool n1,n2,n3,n4;

  if (IntersectProp(a,b,c,d))
    return TRUE;

  else{
    if ( Between(a,b,c) || Between(a,b,d) || Between(c,d,a) || Between(c,d,b))
  return TRUE;
else
  return FALSE;
  }
}
*****************************************************************

Bool IntersectProp( tPointi a, tPointi b, tPointi c, tPointi d)
{
  if ( Collinear(a,b,c) || Collinear(a,b,d)
       || Collinear(c,d,a) || Collinear(c,d,b))
    return FALSE;

  return Xor( Left(a,b,c), Left(a,b,d)) && Xor( Left(c,d,a), Left(c,d,b));
}
*****************************************************************
```

```
#--------------------------------------------------------------------#
# HMAlgor, supprimer les diagonales nonessentielles. J'ajoute        #
# quelleques choses pour enregistrer les souspolygones, avec le valeur#
# de retour comme le nombre de souspolygone.                         #
#                                                                    #
# Les souspolygones sont enregistrés dans poly[i], ième polygone     #
# convexe, avec ns[i], le nombre de sommet.                          #
#--------------------------------------------------------------------#

int HMAlgor(tPolygoni *poly, int ns[], DIAGONAL_STRUCT
            *PolygonDiagonals, POLYGON_STRUCT *PolygonVertices,
             int nvertices, int ndiagonals)
{
   ////////////////////////////////////////////////////////////////////
   //
   // go from 0 to ndiagonals of PolygonDiagonals[ ]
   // examine both endpoints check if convex (convexfrom and convexto)
   // if both true,    then both ends are convex
   //                          -- can be removed (set exist = FALSE)
   // if either false, then at least one is reflex -- do not remove
   //
   // print out the following in PostScript format
   // (1) original polygon using PolygonVertices[ ]
   // (2) the new "essential" diagonals (check if exist = TRUE)
   //
   ////////////////////////////////////////////////////////////////////

   int i,k;           // for loop counter
   double x, y;     // temp x and y coordinates
   int dfrom, dto;  // temp diagonal from/to endpoints
   int d,j;
   int index[MAX_VERTICES];
   int vdiag[MAX_VERTICES];
   int np, nv;          // index of the convex polygon; nb of the vertices
   int max, ind, temp;
   DiagonalTest Diag[MAX_VERTICES];

   for (i = 0; i < ndiagonals; i++)  // remove "inessential" diagonals
   {

      if (PolygonDiagonals[i].convexfrom && PolygonDiagonals[i].convexto)
         PolygonDiagonals[i].exist = FALSE;

   } // end of for

   // Draw the original polygon

   printf("\n%%Polygon:\n");
   printf("oldpath\n");
```

```
x = PolygonVertices[0].xv; y = PolygonVertices[0].yv;
printf("%8.2lf\t%8.2lf\tmoveto\n", x, y);

for (i = 1; i < nvertices; i++)
{
   x = PolygonVertices[i].xv;  y = PolygonVertices[i].yv;
   printf("%8.2lf\t%8.2lf\tlineto\n", x, y);
} // end of for

printf("closepath stroke\n");

// Draw the diagonals (with inessential removed)

printf("\nnewpath\n");
j = 0; // number of essential diagonals

for (i = 0; i < ndiagonals; i++)
{
   if (PolygonDiagonals[i].exist)  // does diagonal still exist?
   {
     dfrom = PolygonDiagonals[i].vfrom;  // if yes get from/to endpoints
     dto   = PolygonDiagonals[i].vto;
     printf("%%Diagonal: (%d,%d)\n", dfrom, dto);

     Diag[j].from = dfrom;
     Diag[j].to = dto;
     Diag[j].use = FALSE;
     Diag[j+1].from = dto;
     Diag[j+1].to = dfrom;
     Diag[j+1].use = FALSE;
     j+=2;

      // get x and y coord of from/to diagonal endpoints

      x = PolygonVertices[dfrom].xv;  y = PolygonVertices[dfrom].yv;
      printf("%8.2lf\t%8.2lf\tmoveto\n", x, y);

      x = PolygonVertices[dto].xv;    y = PolygonVertices[dto].yv;
      printf("%8.2lf\t%8.2lf\tlineto\n", x, y);

   }

} // end of for
d = j;
printf("closepath stroke\n\n");

vdiag[0]=Diag[0].from;
vdiag[1]=Diag[0].to;
k = 2;
for ( i=1; i<d; i=i+2)
```

```
  {
    j = 0;
    do {
      if ( Diag[i].from==vdiag[j])
        break;
      j++;
    } while(j<k);
    if (j==k)
    {
      vdiag[k] = Diag[i].from;
      k++;
    }

    j = 0;
    do {
      if ( Diag[i].to==vdiag[j])
        break;
      j++;
    } while(j<k);
    if (j==k)
    {
      vdiag[k] = Diag[i].to;
      k++;
    }
  }
nv = k;

for (i=0;i<nv;i++)
  for (j=0;j<i;j++)
    if(vdiag[i]>vdiag[j])
      {
        temp = vdiag[i];
        vdiag[i] = vdiag[j];
        vdiag[j] = temp;
      }
// vdiag[ ] dans l'ordre de diminuer
vdiag[nv] = vdiag[0];

// save the convex polygons
np = 0;               // np: nombre de polygone

for (i=0;i<d;i++)     // d: nombre de  diagonal qui reste

    for (j=0;j<=nv-1;j++)    // nv: vombre de sommet qui apparait
                             // dans les diagonals qui restent
      if(Diag[i].from==vdiag[j] && Diag[i].to==vdiag[j+1]
                               && Diag[i].use==FALSE)
        {
          index[0] = Diag[i].from;
          index[1] = Diag[i].to;
```

```
            Diag[i].use = TRUE;
            j = 2;
            do {
              if(index[j-1]!=nvertices-1)
                index[j] = index[j-1]+1;
              else
                index[j] = 0;
              j++;
            } while(index[j-1]!=index[0]);

            printf("%%%dth polygon\n",np+1);
            for(k=0;k<j-1;k++)
              {
                printf(" %d - ", index[k]);
                poly[np][k][X] = PolygonVertices[index[k]].xv;
                poly[np][k][Y] = PolygonVertices[index[k]].yv;
              }
            printf("\n");
            ns[np] = j-1;     // ns: nombre de sommet pour le polygone convexe n°np
            np++;
          }
    // enregistrer les polygones qui sont au bout

newpoly:
  for ( i=0; i<d; i++ )

      if ( Diag[i].use==FALSE )
        {
          index[0] = Diag[i].from;
          index[1] = Diag[i].to;
          Diag[i].use = TRUE;
          j = 2;

          do {
            max = 1;     // pas pour calculer le prochain index
            for(k=0; k<d; k++)
              if ( !Diag[k].use && Diag[k].from==index[j-1]
                                && Diag[k].to!=index[j-2]
                                && abs(Diag[k].to-Diag[k].from)>max
                                && Diag[k].from!=index[0]-1)
                 {
                  max = Diag[k].to - Diag[k].from;
                  ind = k;
                  if ( Diag[k].to == index[0])
                   goto ass;  // on cherche le sommet qui a le lien
                              // avec index[j-1]  et qui a l'index
                              // le plus loin de index[j-1], si le
                              // prohain sommet n'a pas index[0]
                 }
          ass:
```

```
              // assignation
             if (index[j-1] != nvertices-1 )
             // si index[j-1] n'est pas le dernier, soit on obtient
             // l'index par ajouter 1, soit utiliser le diagonal
             // essentiel qui n'est pas encore utilisé
              {
                index[j] = index[j-1] + max;
                if ( max != 1 )
                  {
                    Diag[ind].use = TRUE;
   // si il est un des diagonals essentiels, on marke qu'il est utilisé.
                    max = 1;
                  }
                j++;
               }
              else  // si index[j-1] est le dernier, on considère le cycle
              {
                if (Diag[k].to==index[0])
                 {
                    index[j] = index[j-1] + max;
                    Diag[k].use = TRUE;
                  }
                else
                  index[j] = 0;
                j++;
              }
          } while (index[j-1] != index[0]);

          printf("%dth polygon\n", np+1);

         for(k=0;k<j-1;k++)
           {
              printf(" %d - ", index[k]);
              poly[np][k][X] = PolygonVertices[index[k]].xv;
              poly[np][k][Y] = PolygonVertices[index[k]].yv;
            }
          printf("\n");
          ns[np] = j-1;
          np++;
          goto newpoly;
          // cherche si il y a encore des diagonals essentiels nonutilisés
        }

   return np;
} // end of HMAlgor
```

A.6 Calcul de l'intégration

(Dans le fichier "intflux.c")

```
#-------------------------------------------------------------------------#
# Après on donne les pas d'intégration, nous pouvons calculer             #
# l'intégration sur les polygones convexes.                               #
#                                                                         #
# L'idée: on construit un rectangle qui comprit la somme de Minkowski,    #
# on considère les points dans ce rectangle, et teste si ce point est     #
# dans la somme de Minkowski. On ne fait l'intégration que dans le cas    #
# que InPolyConvex est vrai.                                              #
#-------------------------------------------------------------------------#

double Integration (tPolygoni Ai, tPolygoni Bi, int ai, int bi,
                    double h, double l)
{
  int i,j,c;
  int ixmax, ixmin, iymax, iymin;
  int k;
  Bool w,z;
  double sa, area;
  double r, x, y; // valeurs aléatoires
  tPointd T1,t;
  tPolygond D;
  double rst;
  tdVertex pointer, ipointer;
  rst = 0;

  k = SommeMinkowski(Ai, ai, Bi, bi );

  i = 0;
  ixmax = sommeM[i][X];
  ixmin = sommeM[i][X];
  iymax = sommeM[i][Y];
  iymin = sommeM[i][Y];

  do {
    i = i+1;
    if (sommeM[i][X] > ixmax)  ixmax = sommeM[i][X];
    else if (sommeM[i][X] < ixmin)  ixmin = sommeM[i][X];
    if (sommeM[i][Y] > iymax)  iymax = sommeM[i][Y];
    else if (sommeM[i][Y] < iymin)  iymin = sommeM[i][Y];
  } while(i<k-1);

  sa = h*l;

  r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)) );
  // r is a random floating point value in the range [0,1)
  // {including 0, not including 1}. Note we must convert rand()
```

```
  // and/or RAND_MAX+1 to floating point values to avoid integer
  // division. In addition, Sean Scanlon pointed out the possibility
  // that RAND_MAX may be the largest positive integer the architecture
  // can represent, so (RAND_MAX+1) may result in an overflow,
  // or more likely the value will end up being the largest negative
  // integer the architecture can represent, so to avoid this we
  // convert RAND_MAX and 1 to doubles before adding.
  x = (r * l);
  r = ((double)rand() / ((double)(RAND_MAX)+(double)(1)) );
  y = (r * h);

  j = 0;
  while ((t[Y] = iymin + j*h + y)<=iymax)  {
    i = 0;
    while ((t[X] = ixmin + i*l + x)<=ixmax) {
      // w = InPolyConvex(t, sommeM, k);
      w = 1;

      if (w)
        {
          for (c=0;c<bi;c++)
            {
              Assd(T1,Bi[c]);
              SubVecd(T1, t, D[c]);
            }

          z = ConvexIntersect(Ai,ai,D,bi);
  if(z)
    {
      area = polygon_area_2()/2;

      pointer = intersection->next;
      do {
ipointer = pointer;
pointer = ipointer->next;
DELETE(intersection, ipointer);
      } while(pointer!=intersection);
      FREE(intersection);
    }
          else   area = 0;

          rst = rst+function(t)*area;
     }
      i++;
    }
    j++;
  }
  return rst*sa;
}
```

A.7 Calcul de la somme de Minkowski

(Dans le fichier “zoneintegration.c”)

```
#-------------------------------------------------------------------#
#                     La somme de Minkowski                         #
# On peut trouver l'algorithme dans le livre << computational     #
# geometry in C >> de Joseph O'Rourke.                              #
#                                                                   #
# Deux polygones A et B, avec ses nombre de sommet ni et si, sont #
# comme les entrées. Comme la sortie, c'est le nombre de sommet de#
# la somme de Minkowski de ces deux polygones.                      #
#-------------------------------------------------------------------#

int SommeMinkowski( tPolygoni A, int ni, tPolygoni B, int si)
{
  tPointi p0 = {0,0};
  int j0;       // index of start point
  int k;
  j0 = ReadPoints( p0, A, ni, B, si );
  Vectorize ();
  qsort(
        &P[0],               // pointer to the first elem
        m,                   // number of elems
        sizeof ( tsPoint),   // size of each elem
        Compare              // -1,0,+1 compare function
        );
  k =  Convolve ( j0, p0 );
  return k;
}
*************************************************************************


// Reads in the coordinates of the points from stdin, filling in P, and
// setting the three global counters m=n+s. Returns a start point p0.

int ReadPoints( tPointi p0, tPolygoni A, int ni, tPolygoni B, int si )
{
   int i;
   int pxmin, pymin, pxmax, pymax; // Primary min & max
   int sxmin, symin, sxmax, symax; // Secondary min & max
   int mp, ms; // i index of max (u-r) primary and secondary points

   m = 0;
   n = ni;
   if ( n > PMAX )
     printf("Error in ReadPoints:  > %d\n", PMAX), exit(EXIT_FAILURE);

   for( i = 0; i < n; i++ ) {
```

```
  P[i].v[X] = - A[i][X];
  P[i].v[Y] = - A[i][Y];
   /* Reflect primary polygon */
  P[i].vnum = i;
  P[i].primary = TRUE;
  m++;
}

s = si;
if ((m=n+s) > PMAX )
  printf("Error in ReadPoints:  > %d\n", PMAX), exit(EXIT_FAILURE);

for( i = 0; i < s; i++ ) {

  P[n+i].v[X] = B[i][X];
  P[n+i].v[Y] = B[i][Y];
  P[n+i].vnum = i;
  P[n+i].primary = FALSE;
  m++;
}

// Compute Bounding Box and output Postscript header.
pxmin = pxmax = P[0].v[X];
pymin = pymax = P[0].v[Y];
mp = 0;
for (i = 1; i < ni; i++) {
   if      ( P[i].v[X] > pxmax ) pxmax = P[i].v[X];
   else if ( P[i].v[X] < pxmin ) pxmin = P[i].v[X];
   if      ( P[i].v[Y] > pymax ) {pymax = P[i].v[Y]; mp = i;}
   else if ( P[i].v[Y] == pymax && (P[i].v[X] > P[mp].v[X]) ) mp = i;
   else if ( P[i].v[Y] < pymin ) pymin = P[i].v[Y];
}

sxmin = sxmax = P[n].v[X];
symin = symax = P[n].v[Y];
ms = n;
for (i = 1; i < s; i++) {
   if      ( P[n+i].v[X] > sxmax ) sxmax = P[n+i].v[X];
   else if ( P[n+i].v[X] < sxmin ) sxmin =  P[n+i].v[X];
   if      ( P[n+i].v[Y] > symax ) {symax =  P[n+i].v[Y]; ms = n+i;}
   else if ( P[n+i].v[Y] == symax && ( P[n+i].v[X] > P[ms].v[X]) ) ms = n+i;
   else if ( P[n+i].v[Y] < symin ) symin = P[n+i].v[Y];
}

// Compute the start point: upper rightmost of both.

AddVec( p0, P[mp].v, p0 );

AddVec( p0, P[ms].v, p0 );
```

```
   return mp; // j0: starting index.
}
*****************************************************************************

void Vectorize ( )
{
  int i;
  tPointi last;

  SubVec( P[0].v, P[n-1].v, last);
  for(i=0;i<n-1;i++)
    SubVec(P[i+1].v, P[i].v, P[i].v);
  P[n-1].v[X] = last[X];
  P[n-1].v[Y] = last[Y];

  SubVec( P[n].v, P[n+s-1].v, last);
  for ( i = 0; i< s-1; i++)
    SubVec( P[n+i+1].v, P[n+i].v, P[n+i].v);
  P[n+s-1].v[X]=last[X];
  P[n+s-1].v[Y]=last[Y];
}
*****************************************************************************

int Compare ( const void *tpi, const void *tpj)
{
  int a;                        // AreaSign result
  int x,y;                      // projections in the first quadrant
  tPoint pi, pj;                // Recasted points
  tPointi Origin = {0,0};
  pi = (tPoint)tpi;
  pj = (tPoint)tpj;
  tPointd T1,T2,T3;

  if ((pi->v[Y] > 0) &&(pj->v[Y] <= 0))
    return 1;
  else if((pi->v[Y] <= 0) &&(pj->v[Y] > 0))
    return -1;

  else if((pi->v[Y] == 0) &&(pj->v[Y] == 0))
    {
      if ((pi->v[X] < 0) &&(pj->v[X] > 0))
        return -1;
      if ((pi->v[X] > 0) &&(pj->v[X] < 0))
        return 1;
      else if ( abs(pi->v[X]) < abs(pj->v[X]))
        return -1;
      else if ( abs(pi->v[X]) > abs(pj->v[X]))
        return 1;
      else
        return 0;
```

```
    }
  else
    {
      Assd(T1,Origin);
      Assd(T2,pi->v);
      Assd(T3,pj->v);
      a = AreaSign (T1, T2, T3);
      if (a>0)
        return -1;
      else if (a<0)
        return 1;
      else
        {
          x = abs ( pi->v[X]) - abs( pj->v[X]);
          y = abs ( pi->v[Y]) - abs( pj->v[Y]);
          if ((x<0)||(y<0))
            return -1;
          else if ( (x>0)||(y>0))
            return 1;
          else
            return 0;
        }
    }
}
*************************************************************************

#-----------------------------------------------------------------#
# Enregristrer les sommets de la somme de Minkowski dans le tableau #
# sommeM[ ], avec le nombre de sommet comme retour                #
#-----------------------------------------------------------------#

int Convolve ( int j0, tPointi p)
{
  int i;       // Index into sorted edge vectors P
  int j;       // Primary polygon index
  int k;
  int l;
  k = 0;

  i = 0;
  j = j0;
  do{
    while( !(P[i].primary && P[i].vnum == j))
      {
 if ( !P[i].primary)
  {
    AddVec(p,P[i].v, p);
     l=0;
     while (l<k) {
       if(p[X]!=sommeM[l][X] || p[Y]!=sommeM[l][Y])
```

```
 l++;
       else break;
     }
     if (l==k)
       {
 Assi(p, sommeM[k]);
 k++;
       }
  }

i = (i+1)%m;
      }
    AddVec( p,P[i].v, p);
    l=0;
    while (l<k) {
      if(p[X]!=sommeM[l][X] || p[Y]!=sommeM[l][Y])
l++;
      else break;
    }
    if (l==k)
      {
Assi(p, sommeM[k]);
k++;
      }
    j=(j+1)%n;
  }while  (j != j0);

  while (i != 0)
    {
      if ( !P[i].primary )
   {
  AddVec(p, P[i].v, p);
  l=0;
  while (l<k) {
    if(p[X]!=sommeM[l][X] || p[Y]!=sommeM[l][Y])
      l++;
    else break;

   }
   if (l==k)
   {
     Assi(p, sommeM[k]);
     k++;
   }
   }
      i = (i+1)%m;
    }
  return k;
}
```

A.8 Tester si un point est à l'intérieur d'un polygone convexe

(Dans le fichier "geom.c")

```
#-------------------------------------------------------------------#
# Pour tester si un point est dans un polygone, on utilise la       #
# fonction LeftOnd(tPointd, tPointd, tPointd), d: double            #
#-------------------------------------------------------------------#

Bool InPolyConvex ( tPointd t, tPolygoni M, int k )
{
  tPointd T1,T2;
  int i;
  i = 0;

  do {
    Assd(T1,M[i]);
    Assd(T2,M[i+1]);
    if (!LeftOnd(T1, T2, t ))
      {
return FALSE;
      }
    i++;
  } while (i<k);

  Assd(T1,M[k-1]);
  Assd(T2,M[0]);
  if (!LeftOnd(T1, T2, t))
    {
      return FALSE;
    }

  return TRUE;
}
```

A.9 Calcul de l'intersection

(Dans le fichier "intersection.c")

```
#-------------------------------------------------------------#
#     Calcul l'intersection de deux polygone convexe.         #
#-------------------------------------------------------------#



Bool    ConvexIntersect( tPolygoni M, int n, tPolygond N, int s )
                          // M has n vertices, N has s vertices.
{
  int     a, b;           // indices on M and N (resp.)
  int     a1, b1;         // a-1, b-1 (resp.)
  tPointi A;
  tPointd B;              // directed edges on M and N (resp.)
  int     cross;          // sign of z-component of A x B
  int     bHA, aHB;       // b in H(A); a in H(b).
  tPointi Origin = {0,0}; // (0,0)
  tPointd p;              // double point of intersection
  tPointd q;              // second point of intersection
  tInFlag inflag;         // {Pin, Qin, Unknown}: which inside
  int     aa, ba;         // # advances on a & b indices (after 1st inter.)
  Bool    FirstPoint;     // Is this the first point? (used to initialize).
  tPointd p0;             // The first point.
  char     code;          // SegSegInt return code.
  tdVertex v;
  int vnum = 0;
  tPointd T1, T2;


  // Initialize variables.
  a = 0; b = 0; aa = 0; ba = 0;
  inflag = Unknown; FirstPoint = TRUE;

  do {
    // Computations of key variables.
    a1 =(a + n - 1) % n;
    b1 =(b + s - 1) % s;
    SubVec( M[a], M[a1], A );
    SubVecd( N[b], N[b1], B );       // tPointd

    Assd(T1, Origin); Assd(T2, A);
    cross = AreaSign(T1,T2,B );
    Assd(T1,M[a]);
    aHB   = AreaSign( N[b1], N[b], T1 );
    Assd(T1,M[a1]); Assd(T2,M[a]);
    bHA   = AreaSign(T1,T2, N[b] );

    // If A & B intersect, update inflag.
```

```
    code = SegSegInt(T1, T2, N[b1], N[b], p, q);
    if ( code == '1' || code == 'v' ) {
      if ( inflag == Unknown && FirstPoint ) {
          aa = ba = 0;
          FirstPoint = FALSE;
          p0[X] = p[X]; p0[Y] = p[Y];
}
      inflag = InOut( p, inflag, aHB, bHA,vnum );
    }

    //-----Advance rules-----

    // Special case: A & B overlap and oppositely oriented.
    Assd(T1,A);
    if ( ( code == 'e' ) && (Dot( T1, B ) < 0) )
      PrintSharedSeg( p, q ), exit(EXIT_SUCCESS);

    // Special case: A & B parallel and separated.
    if ( (cross == 0) && ( aHB < 0) && ( bHA < 0 ) )
      printf("%%P and Q are disjoint.\n"), exit(EXIT_SUCCESS);

    // Special case: A & B collinear.
    else if ( (cross == 0) && ( aHB == 0) && ( bHA == 0 ) ) {
      // Advance but do not output point.
      if ( inflag == Pin )
        b = Advance( b, &ba, s, inflag == Qin, N[b], vnum );
      else
        {
          Assd(T1,M[a]);
          a = Advance( a, &aa, n, inflag == Pin,T1, vnum );
        }
      }

    // Generic cases.
    else if ( cross >= 0 ) {
     if ( bHA > 0)
         {
           Assd(T1,M[a]);
           a = Advance( a, &aa, n, inflag == Pin, T1, vnum );
         }
      else
         b = Advance( b, &ba, s, inflag == Qin, N[b], vnum );
    }
    else {
      if ( aHB > 0)
         b = Advance( b, &ba, s, inflag == Qin, N[b], vnum );
      else
        {
          Assd(T1,M[a]);
          a = Advance( a, &aa, n, inflag == Pin, T1, vnum );
```

```
          }
      }

   // Quit when both adv. indices have cycled, or one has cycled twice.
  } while ( ((aa < n) || (ba < s)) && (aa < 2*n) && (ba < 2*s) );

  if ( !FirstPoint ) // If at least one point output, close up.
     {
       v = MakeNulldVertex ();
       v -> v[X] = p[X];
       v -> v[Y] = p[Y];
       v -> vnum = vnum++;
     }

  // Deal with special cases: not implemented.
  if ( inflag == Unknown)
     return FALSE;

  return TRUE;
}
*****************************************************************************

tInFlag InOut( tPointd p, tInFlag inflag, int aHB, int bHA,int vnum )
{
  tdVertex v;
  v = MakeNulldVertex ();
  v -> v[X] = p[X];
  v -> v[Y] = p[Y];
  v -> vnum = vnum++;

  // Update inflag.
  if      ( aHB > 0)
    return Pin;
  else if ( bHA > 0)
    return Qin;
  else    // Keep status quo.
    return inflag;
}
*****************************************************************************

int     Advance( int a, int *aa, int n, Bool inside, tPointd v, int vnum )
{
  tdVertex d;
  if ( inside )
    {
      d = MakeNulldVertex ();
      d -> v[X] = v[X];
      d -> v[Y] = v[Y];
      d -> vnum = vnum++;
    }
```

```
  (*aa)++;
  return  (a+1) % n;
}
*********************************************************************************

/*-------------------------------------------------------------------
SegSegInt: Finds the point of intersection p between two closed
segments ab and cd.  Returns p and a char with the following meaning:
   'e': The segments collinearly overlap, sharing a point.
   'v': An endpoint (vertex) of one segment is on the other segment,
        but 'e' doesn't hold.
   '1': The segments intersect properly (i.e., they share a point and
        neither 'v' nor 'e' holds).
   '0': The segments do not intersect (i.e., they share no points).
Note that two collinear segments that share just one point, an endpoint
of each, returns 'e' rather than 'v' as one might expect.
-------------------------------------------------------------------*/
char SegSegInt( tPointd a, tPointd b, tPointd c, tPointd d,
                tPointd p, tPointd q)
{
   double  s, t;       // The two parameters of the parametric eqns.
   double num, denom;  // Numerator and denoninator of equations.
   char code = '?';    // Return char characterizing intersection.

   denom = a[X] * ( d[Y] - c[Y] ) +
           b[X] * ( c[Y] - d[Y] ) +
           d[X] * ( b[Y] - a[Y] ) +
           c[X] * ( a[Y] - b[Y] );

   // If denom is zero, then segments are parallel: handle separately.
   if (denom == 0.0)
      return  ParallelInt(a, b, c, d, p, q);

   num =    a[X] * ( d[Y] - c[Y] ) +
            c[X] * ( a[Y] - d[Y] ) +
            d[X] * ( c[Y] - a[Y] );
   if ( (num == 0.0) || (num == denom) ) code = 'v';
   s = num / denom;

   num = -( a[X] * ( c[Y] - b[Y] ) +
            b[X] * ( a[Y] - c[Y] ) +
            c[X] * ( b[Y] - a[Y] ) );
   if ( (num == 0.0) || (num == denom) ) code = 'v';
   t = num / denom;

   if      ( (0.0 < s) && (s < 1.0) &&
             (0.0 < t) && (t < 1.0) )
     code = '1';
   else if ( (0.0 > s) || (s > 1.0) ||
             (0.0 > t) || (t > 1.0) )
```

```
     code = '0';

   p[X] = a[X] + s * ( b[X] - a[X] );
   p[Y] = a[Y] + s * ( b[Y] - a[Y] );

   return code;
}
*****************************************************************************

double  Dot( tPointd a, tPointd b )
{
    int i;
    double sum = 0.0;

    for( i = 0; i < DIM; i++ )
       sum += a[i] * b[i];

    return  sum;
}
*****************************************************************************

void PrintSharedSeg( tPointd p, tPointd q )
{
   printf("%%A int B:\n");
   printf("%8.2lf %8.2lf moveto\n", p[X], p[Y] );
   printf("%8.2lf %8.2lf lineto\n", q[X], q[Y] );
   ClosePostscript();
}
*****************************************************************************

void   ClosePostscript( void )
{
   printf("closepath stroke\n");
   printf("showpage\n%%%%EOF\n");
}
*****************************************************************************

char ParallelInt( tPointd a, tPointd b, tPointd c, tPointd d,
                  tPointd p, tPointd q )
{
   if ( !Collineard( a, b, c) )
      return '0';

   if ( Betweend( a, b, c ) && Betweend( a, b, d ) ) {
       Ass( p, c );
       Ass( q, d );
       return 'e';
   }
   if ( Betweend( c, d, a ) && Betweend( c, d, b ) ) {
     Ass( p, a );
```

```
    Ass( q, b );
    return 'e';
  }
  if ( Betweend( a, b, c ) && Betweend( c, d, b ) ) {
    Ass( p, c );
    Ass( q, b );
    return 'e';
  }
  if ( Betweend( a, b, c ) && Betweend( c, d, a ) ) {
    Ass( p, c );
    Ass( q, a );
    return 'e';
  }
  if ( Betweend( a, b, d ) && Betweend( c, d, b ) ) {
    Ass( p, d );
    Ass( q, b );
    return 'e';
  }
  if ( Betweend( a, b, d ) && Betweend( c, d, a ) ) {
     Ass( p, d );
     Ass( q, a );
     return 'e';
  }
  return '0';
}
```

A.10 Autres

(Dans le fichier “intflux.h”)

```
#-------------------------------------------------------------------------#
#            Les autres définitions et déclarations                       #
#-------------------------------------------------------------------------#

// Define Boolean type
typedef enum { FALSE, TRUE } Bool;
typedef enum { Pin, Qin, Unknown } tInFlag;

#define EXIT_FAILURE   1
// Define vertex indices.
#define X   0
#define Y   1

#define DIM 2
typedef int tPointi[DIM];   // Type int point
typedef double tPointd[DIM];

// Define structures for vertices, edges
typedef struct tVertexStructure tsVertex;
typedef tsVertex *tVertex;

typedef struct tdVertexStructure tdsVertex;
typedef tdsVertex *tdVertex;


struct tVertexStructure {
  tPointi  v;
  int    vnum;
  Bool     ear;                 // T iff an ear
  tVertex  next, prev;
};

struct tdVertexStructure {
  tPointd  v;
  int    vnum;
  tdVertex  next, prev;
};

/* Define flags */
#define SAFE 1000000 /* Range of safe coord values. */

/* Global variable definitions */

tdVertex intersection;

typedef struct tPointStructure tsPoint;
```

```
typedef tsPoint *tPoint;
struct tPointStructure {
  int vnum;
  tPointi v;
  Bool primary;
};

#define MAX 80
#define MAX_VERTICES  80
#define PMAX 1000
typedef tsPoint tPointArray[PMAX];


typedef tPointi tPolygoni[PMAX];  // type integer polygon
typedef tPointd tPolygond[PMAX];


typedef struct      // struct for saving diagonals in array
{
   Bool exist;      // whether diagonal exist (essential/non)
   int  vfrom;      // index from vertex of diagonal endpoint
   Bool convexfrom; // whether from is convex
   int  vto;        // index to vertex of diagonal endpoint
   Bool convexto;   // whether to is convex

} DIAGONAL_STRUCT;


typedef struct // struct for saving vertices as array
{
   int xv;     // x coordinate
   int yv;     // y coordinate

} POLYGON_STRUCT;


typedef struct   // structure for the intermediate diagonal
{
  Bool use;        // TRUE iff used
  int from;        // index from vertex of diagonal endpoint
  int to;          // index to vertex of diagonal endpoint
} DiagonalTest;


// NEW, ADD and DELETE macros

#define NEW(p, type)\
  if ((p=(type *) malloc (sizeof(type)))==NULL) {\
  printf ("NEW: Out of Memory!\n");\
  exit(EXIT_FAILURE);\
```

```
}

#define ADD(head, p) if(head){\
  p->prev = head;\
  p->next = head->next;\
  head->next = p;\
  p->next->prev = p;\
}\
else {\
  head = p;\
  head->next = head->prev = p;\
}

#define ADDP(head,p) if(head){\
  p->next = head;\
  p->prev = head->prev;\
  head->prev = p;\
  p->prev->next = p;\
}\
else {\
  head = p;\
  head->next = head->prev = p;\
}

#define FREE(p)  if(p) {free((char *)p); p = NULL;}

tPolygoni sommeM;

int     SommeMinkowski ( tPolygoni A, int ni, tPolygoni B, int si );
void    Assd (tPointd B,tPointi A);
void    SubVecd( tPointd a, tPointd b, tPointd c );
Bool    ConvexIntersect( tPolygoni M, int n, tPolygond N, int s);
double  polygon_area_2(void);
int     ReadVertices( int i, POLYGON_STRUCT *PolygonVertices,
                    tVertex vertices, FILE *fp );
int     Area2i ( tPointi a, tPointi b, tPointi c );
Bool    Convexity ( tVertex vertices );
int     Triangulate( DIAGONAL_STRUCT *PolygonDiagonals,tVertex vertices, int nvertices );
int     HMAlgor( tPolygoni *poly, int ns[],DIAGONAL_STRUCT *PolygonDiagonals,
                 POLYGON_STRUCT *PolygonVertices, int nvertices, int ndiagonals );
void    AddVec( tPointi a, tPointi b, tPointi c );
void    SubVec( tPointi a, tPointi b, tPointi c );
int     AreaSign ( tPointd a, tPointd b, tPointd c );
void    Assi ( tPointi a, tPointi b );
tdVertex MakeNulldVertex(void);
Bool    Collineard( tPointd a, tPointd b, tPointd c);
Bool    Between( tPointi a, tPointi b, tPointi c);
void    Ass (tPointd B,tPointd A);
Bool    Betweend( tPointd a, tPointd b, tPointd c);
Bool    InPolyConvex( tPointd t, tPolygoni M, int k);
```

(Dans le fichier “geom.c”)

```
Bool Left( tPointi a, tPointi b, tPointi c)
{
  return Area2i(a, b, c) > 0;
}

Bool LeftOn( tPointi a, tPointi b, tPointi c)
{
  return Area2i(a, b, c) >= 0;
}
Bool LeftOnd( tPointd a, tPointd b, tPointd c)
{
  return Area2(a, b, c) >= 0;
}

Bool Collinear( tPointi a, tPointi b, tPointi c)
{
  return Area2i(a, b, c) == 0;
}
Bool Collineard( tPointd a, tPointd b, tPointd c)
{
  return Area2(a, b, c) == 0;
}
Bool Xor( Bool x, Bool y)
{
  return !x ^ !y;
}
Bool Between( tPointi a, tPointi b, tPointi c)
{
  tPointi ba, ca;

  if (!Collinear(a,b,c))
    return FALSE;

  if ( a[X] != b[X] )
    return ((a[X] <= c[X]) && (c[X] <= b[X]))
             || ((a[X] >= c[X]) && (c[X] >= b[X]));
  else
    return ((a[Y] <= c[Y]) && (c[Y] <= b[Y]))
             || ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
Bool Betweend( tPointd a, tPointd b, tPointd c)
{
  tPointd ba, ca;

  if (!Collineard(a,b,c))
    return FALSE;
```

```
  if ( a[X] != b[X] )
    return ((a[X] <= c[X]) && (c[X] <= b[X]))
             || ((a[X] >= c[X]) && (c[X] >= b[X]));
  else
    return ((a[Y] <= c[Y]) && (c[Y] <= b[Y]))
             || ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
double Area2 ( tPointd a, tPointd b, tPointd c)
{
  return (b[X] - a[X])*(c[Y]-a[Y]) - (c[X] - a[X])*(b[Y]-a[Y]);
}

int Area2i ( tPointi a, tPointi b, tPointi c)
{
  return (b[X] - a[X])*(c[Y]-a[Y]) - (c[X] - a[X])*(b[Y]-a[Y]);
}

int AreaSign ( tPointd a, tPointd b, tPointd c )
{
  double area2;

  area2 = ( b[0] - a[0] ) * ( c[1] - a[1] ) -
          ( c[0] - a[0] ) * ( b[1] - a[1] );

  if       ( area2 >  0.5 )   return 1;
  else if  ( area2 < -0.5 )   return -1;
  else                        return 0;
}


// Area for a polygon
double polygon_area_2()
{
  tdVertex  p;
  double sum = 0;

  p = intersection->next;
  do {
    sum = sum + Area2(intersection->v, p->v, p->next->v );

     p = p->next;
  } while (p->next != intersection);
  return  sum;
}

void Assd (tPointd B,tPointi A)
{
  B[X] = A[X];
  B[Y] = A[Y];
}
```

```
void Assi ( tPointi a, tPointi b)
{
  int i;
  for ( i=0; i<DIM; i++ )
    b[i] = a[i];
}
// Subtraction of points (vector)
void SubVec( tPointi a, tPointi b, tPointi c )
{
  int i;
  // a-b->c
  for ( i = 0; i<DIM; i++)
    c[i]=a[i]-b[i];
}

void SubVecd( tPointd a, tPointd b, tPointd c )
{
  int i;
  // a-b->c
  for ( i = 0; i<DIM; i++)
    c[i]=a[i]-b[i];
}

// Addition of points (vector)
void AddVec( tPointi a, tPointi b, tPointi c)
{
  int i;

  for (i=0; i<DIM; i++)
    c[i] = a[i] + b[i];
}
void AddVecd( tPointd a, tPointd b, tPointd c)
{
  int i;

  for (i=0; i<DIM; i++)
    c[i] = a[i] + b[i];
}

tVertex MakeNullVertex( tVertex vertices )
{
   tVertex  v;

   NEW( v, tsVertex );
   ADD( vertices, v );

   return v;
}
tdVertex MakeNulldVertex( void )
{
```

```
   tdVertex  v;

   NEW( v, tdsVertex );
   ADDP( intersection, v );

   return v;
}
```

Annexe B

Exemple de fichier pour les coordonnées de polygones

« poly_coord.txt »

```
ID      xcoord          ycoord
1       540116.0000     1795000.0000
1       540261.0000     1795000.0000
1       540274.0000     1794920.0000
1       540139.0000     1794900.0000
1       540116.0000     1795000.0000
2       540378.0000     1795000.0000
2       540467.0000     1795000.0000
2       540453.0000     1794850.0000
2       540373.0000     1794850.0000
2       540374.0000     1794890.0000
2       540378.0000     1795000.0000
.       .               .
.       .               .
.       .               .

65      540753.0000     1794180.0000
65      540772.0000     1794240.0000
66      540139.0000     1794900.0000
66      540274.0000     1794920.0000
66      540272.0000     1794890.0000
66      540247.0000     1794870.0000
66      540151.0000     1794840.0000
66      540146.0000     1794870.0000
66      540139.0000     1794900.0000
```

Annexe C

Exemple de session

La méthode d'intégration par grille :

Lancement du programme

./intflux ↙

Premières sorties : lecture des polygones, tests de convexité, décompositions éventuelles en sous-polygones convexes

npoly = 66
0 540139 1794900
1 540274 1794920
2 540261 1795000
3 540116 1795000
%Bounding box :
xmax = 540274; xmin = 540116; difference : 158
ymax = 1795000; ymin = 1794900; difference : 100
1 th polygon is convex
****
0 540378 1795000
1 540374 1794890
2 540373 1794850
3 540453 1794850
4 540467 1795000
%Bounding box :
xmax = 540467; xmin = 540373; difference : 94
ymax = 1795000; ymin = 1794850; difference : 150
2 th polygon is convex
****
0 540272 1794890
1 540247 1794870
2 540151 1794840
3 540156 1794820
4 540278 1794800
5 540274 1794850

%Bounding box :
xmax = 540278; xmin = 540151; difference : 127
ymax = 1794890; ymin = 1794800; difference : 90
3 th polygon is nonconvex

%Polygon :
oldpath
540272.00 1794890.00 moveto
540247.00 1794870.00 lineto
540151.00 1794840.00 lineto
540156.00 1794820.00 lineto
540278.00 1794800.00 lineto
540274.00 1794850.00 lineto
closepath stroke

newpath
%Diagonal : (5,1)
540274.00 1794850.00 moveto
540247.00 1794870.00 lineto
closepath stroke

%1th polygon
5 - 1 - 2 - 3 - 4 -
%2th polygon
1 - 5 - 0 -
****

. . .

****
0 540139 1794900
1 540146 1794870
2 540151 1794840
3 540247 1794870
4 540272 1794890
5 540274 1794920
%Bounding box :
xmax = 540274; xmin = 540139; difference : 135
ymax = 1794920; ymin = 1794840; difference : 80
66 th polygon is nonconvex

%Polygon :
oldpath
540139.00 1794900.00 moveto
540146.00 1794870.00 lineto
540151.00 1794840.00 lineto
540247.00 1794870.00 lineto
540272.00 1794890.00 lineto
540274.00 1794920.00 lineto
closepath stroke

newpath
%Diagonal : (4,1)

540272.00 1794890.00 moveto
540146.00 1794870.00 lineto
closepath stroke

%1th polygon
4 - 1 - 2 - 3 -
%2th polygon
1 - 4 - 5 - 0 -
****

Suite du programme : dialogue avec l'utilisateur pour préciser les options

Choose the one you want :
1. you want to calcule the integral of two parcels ;
2. you want to calcule the integrals of all the pairs of parcels.
1↙
Case 1 :
the first polygone : 1↙
the second polygone : 1↙
step for integration x axis : 0.25↙
step for intergration y axis : 0.25↙
how many estimations do you want ?20↙

Suite des sorties : valeurs calculées pour chaque répétition

result[1][1] = 12366
result[1][2] = 12370
result[1][3] = 12374
result[1][4] = 12366
result[1][5] = 12364
result[1][6] = 12362
result[1][7] = 12362
result[1][8] = 12381
result[1][9] = 12364
result[1][10] = 12367
result[1][11] = 12373
result[1][12] = 12372
result[1][13] = 12366
result[1][14] = 12376
result[1][15] = 12374
result[1][16] = 12365
result[1][17] = 12363
result[1][18] = 12362
result[1][19] = 12367
result[1][20] = 12367

Fin des sorties : résultats finaux

result moyenne for the polygones n° 1, 1 = 12368
area * area = 1.63328e+08
standard deviation = 5.13874
coefficient of variation = 0.000415483
Elapsed Time : 83290 milliseconds

La méthode d'intégration par Cubpack++ :

Lancement du programme

./tst ↙

Premières sorties : lecture des polygones, tests de convexité, décompositions éventuelles en sous-polygones convexes

npoly = 66
0 540139 1794900
1 540274 1794920
2 540261 1795000
3 540116 1795000
%Bounding box :
xmax = 540274; xmin = 540116; difference : 158
ymax = 1795000; ymin = 1794900; difference : 100
1 th polygon is convex
****

0 540378 1795000
1 540374 1794890
2 540373 1794850
3 540453 1794850
4 540467 1795000
%Bounding box :
xmax = 540467; xmin = 540373; difference : 94
ymax = 1795000; ymin = 1794850; difference : 150
2 th polygon is convex
****

0 540272 1794890
1 540247 1794870
2 540151 1794840
3 540156 1794820
4 540278 1794800
5 540274 1794850
%Bounding box :
xmax = 540278; xmin = 540151; difference : 127
ymax = 1794890; ymin = 1794800; difference : 90
3 th polygon is nonconvex

%Polygon :
oldpath

540272.00 1794890.00 moveto
540247.00 1794870.00 lineto
540151.00 1794840.00 lineto
540156.00 1794820.00 lineto
540278.00 1794800.00 lineto
540274.00 1794850.00 lineto
closepath stroke

newpath
%Diagonal : (5,1)
540274.00 1794850.00 moveto
540247.00 1794870.00 lineto
closepath stroke

%1th polygon
5 - 1 - 2 - 3 - 4 -
%2th polygon
1 - 5 - 0 -
****

. . .

****
0 540139 1794900
1 540146 1794870
2 540151 1794840
3 540247 1794870
4 540272 1794890
5 540274 1794920
%Bounding box :
xmax = 540274; xmin = 540139; difference : 135
ymax = 1794920; ymin = 1794840; difference : 80
66 th polygon is nonconvex

%Polygon :
oldpath
540139.00 1794900.00 moveto
540146.00 1794870.00 lineto
540151.00 1794840.00 lineto
540247.00 1794870.00 lineto
540272.00 1794890.00 lineto
540274.00 1794920.00 lineto
closepath stroke

newpath
%Diagonal : (4,1)
540272.00 1794890.00 moveto
540146.00 1794870.00 lineto
closepath stroke

%1th polygon
4 - 1 - 2 - 3 -
%2th polygon
1 - 4 - 5 - 0 - **********

Suite du programme : dialogue avec l'utilisateur pour préciser les options

This program is in using Cubpack++
Choose the one you want :
1. you want to calcule the integral of two parcels ;
2. you want to calcule the integrals of all the pairs of parcels.
1↙
Case 1 :
the first polygone : 1↙
the second polygone : 1↙
the precision : 0.0002360↙

Fin des sorties : résultats finaux

precision = 0.000236
result moyenne for the polygones n° 1, 1 = 12367
area * area = 1.63328e+08
confidence interval is [12363.8, 12369.7]
Elapsed Time : 1690 milliseconds