Projet 2A : La chambre la moins froide

Tanguy VIVIER,
Rémi MOUZAYEK

June 1, 2018

Introduction

L'équation de la chaleur est une équation aux dérivées partielles parabolique, pour décrire le phénomène physique de conduction thermique, introduite initialement en 1811 par Jean Baptiste Joseph Fourier. Cette problématique de la diffusion de la chaleur est un enjeu industriel central vos à vis la gestion de l'énergie. A l'heure de la COP 21, elle n'a d'ailleurs jamais semblée autant d'actualité.
Comment maximiser la température d'une chambre en ne jouant que sur la géométrie de cette dernière ? Cette question qui peut surprendre de prime abord - car des dogmes architecturaux nous pousse naïvement à supposer, par définition, que toute chambre respectable se doit d'être rectangulaire - prend alors tout son sens. Cette recherche se place indéniablement dans un vaste domaine d'étude, au confluent des mathématiques et de l'informatique: l'optimisation de forme.

Part I
Mise en place de l'étude

Notre étude sera réalisée en deux dimensions, dans le plan $\mathbb{R}^2$. Par ailleurs, une chambre sera modélisée par un polygone. Nous allons principalement nous intéresser aux pentagones (polygones à 5 côtés), cependant d'autres polygones seront également testés pour mener à bien ce travail.
Cette étude n'est rien d'autre qu'un problème d'optimisation. En effet, nous chercherons à maximiser le critère: température moyenne d'une pièce dans **son régime stationnaire**.
Pour fixer les notations, à l'avenir, la température à la position repérée par le vecteur x de la chambre Ω au temps t sera notée $u(x,t)$. De plus, la température moyenne de la pièce s'exprime facilement par $\frac{1}{|\Omega|}\int_\Omega u(x,t)dx$ ou $|\Omega|$ représente l'aire de la chambre.
Par ailleurs, nous considérerons que la température dans la pièce Ω est pilotée par l'équation aux dérivées partielles:

$$\frac{\partial u}{\partial t}(x,t) = \Delta u(x,t) + f(x,t)$$

Il s'agit de l'équation de la chaleur usuelle dans laquelle les diverses constantes caractéristiques du phénomène ont été égalisé à 1. Par ailleurs, $f(x,t)$ est un terme de source permettant de préciser le profil thermique de la pièce. Une des données clef de ce problème est de bien comprendre que dés lors que nous souhaitons classer des polygones (ou plus généralement des formes géométriques quelconques) il faut nécessairement que nous fixions des caractéristiques. En effet comment comparer la température moyenne de deux pièces si pour un même radiateur elles sont d'aires ou de nombre de cotés différents ?
Voici le cahier des charges retenus:
- le nombre de sommets du polygone est fixé égal à 5 dans l'étude principale, c'est à dire que nous travaillons sur des pentagones;
- l'aire des polygones est fixée égale à $|\Omega|$;
- toutes les chambres seront chauffées grâce à un même radiateur. Pour le modéliser, nous considérerons qu'un mur (c'est à dire un coté du polygone) de longueur fixée l est sujet à un flux thermique entrant Φe connu et, également fixé;
- la chambre est supposée isolée: les températures surfaciques des autres murs (cotés du polygone) sont fixées égales à une même constante T_0;
- Nous considérons un profil thermique identiquement nul : $f(x,t) = 0$.
Le problème est désormais bien posé. Il faut cependant garder à l'esprit que les contraintes fixées ici sont fortes: voyager à aire constant avec la position de deux sommets données est particulièrement contraignant. Une part importante de l'étude est de réfléchir à l'implémentation

d'un algorithme pour pouvoir générer des polygones respectant ces contraintes.

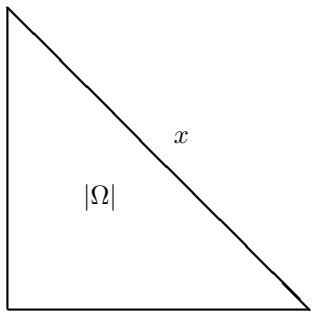
Part II
Étude préliminaire: le cas des triangles

I - Cadre général de l'étude

On va considérer le problème simple d'une pièce triangulaire dans toute cette partie. On considère aussi une application T_S qui a une valeur x va associer un triangle d'aire S et de côtés de longueur x et 1. La fonction u_Ω correspond à la solution de l'équation précédente sur le domaine Ω, qui est dans le cas ici présent un triangle. On va noter M la fonction qui renvoie la moyenne de u_Ω sur Ω telle que $M(\Omega) = \frac{1}{|\Omega|}\int_\Omega u_\Omega(x,t)dx$.
Désormais, on pose $F = M \circ T$. Le problème formulé ci-dessus revient donc à chercher le x qui maximise la fonction F, on se ramène ainsi à un problème d'optimisation de $\mathbb{R}$ dans $\mathbb{R}$.

Pour étudier le problème dans un premier temps, on va considérer un côté fixe de longueur 1 centré sur l'axe des ordonnées. Ensuite on va faire varier le côté adjacent de longueur x. En sachant que l'aire du triangle est fixée, on pourra déterminer le triangle correspondant en fonction du seul paramètre x. On note E l'ensemble des sous-ensembles de $\mathbb{R}^2$

$T_\Omega : \mathbb{R} \longmapsto E$
$x \longrightarrow$ Triangle de côté 1 et x, d'aire $|\Omega|$



Dans la suite de l'étude, on va considérer le côté fixé de longueur 1 sur l'axe des ordonnées, centré sur l'origine (C'est à dire les deux premiers points du triangle en (0, 0.5) et (0, -0.5)). Le dernier point du triangle peut ainsi être déterminé grâce à la seule connaissance de x. Par soucis de simplification, on va ainsi poser la fonction T qui renvoie les coordonnées du dernier point en fonction de x. Le triangle pourra ainsi être tracé car on connait les deux autres coordonnées. Ce dernier point est de coordonnées : $(2|\Omega|, \frac{1}{2} - x\sqrt{1 - (\frac{2|\Omega|}{x})^2})$. (On peut déterminer ces formules à l'aide de calculs simples sur les relations d'Al-Kashi)

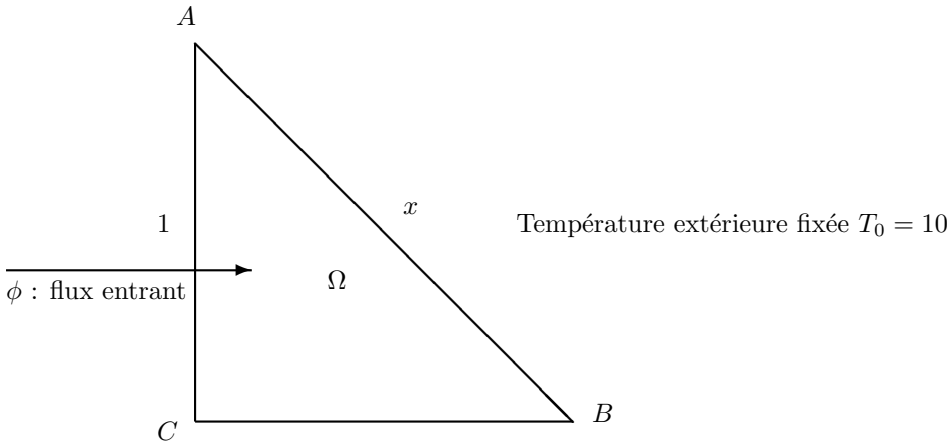
Sur Matlab, on a implémenté une fonction qui construit un objet de type "geometry" triangulaire en fonction de l'aire Ω et de la longueur du côté x. Cet objet geometry définit la forme de la zone sur laquelle on va rédoudre l'équation différentielle.

Listing 1: Construction d'une géométrie Triangulaire

```
function g = triangle(X,area)

% TRIANGLE
% Renvoie la geometrie d'un triangle d'aire fixee et de longueurs de cote
% X et Y

if (X.^2<0.25)
    g = "Un tel triangle n'existe pas"
else
    mat = [2;3;0;0;2*area;-0.5;0.5;0.5-X*(1-(2*area/X)^2)^.5] ;
    [dl,bt] = decsg(mat) ;
    g = dl ;
end
```

II - Première étude : Conditions simples

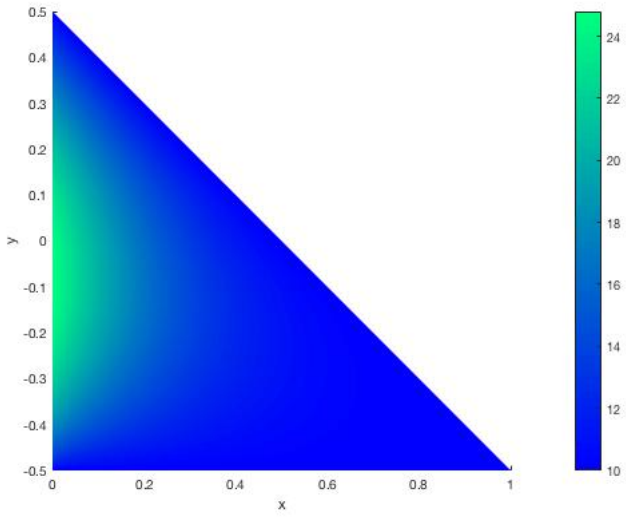
On va considérer ici des conditions relativement simples. Premièrement, on se place en régime stationnaire, c'est à dire que le terme $\frac{\partial u}{\partial t}$ est nul. De plus, on fixe une fonction de chauffage nulle. Enfin, les conditions aux limites sont fixées telles que le côté de longueur fixée 1 soit traversé par un flux positif (modélisant un chauffage), et les deux autre côtés sont fixés à une température extérieure T_0 (dans le cas présent fixé à 10).



En clair les conditions aux limites sont de la forme :

$$\begin{cases} \text{Dirichlet sur } (AB) \cup (CB) \text{: } u_\Omega = 10 \\ \text{Neumann sur } (AC) \text{: } -grad(u_\Omega).n = 50 \end{cases}$$

On peut résoudre l'EDP sur l'exemple ci-dessus avec Matlab assez facilement. On prend dans cette résolution la valeur x $=\sqrt{2}$ et $|\Omega| = 1/2$ pour que cela corresponde au dessin ci-dessus.



Voici le script matlab correspondant (Les fonctions seront explicitées ultérieurement) :

Listing 2: Resolution d'une EDP sur un triangle

```
function result = uTriangle(X,area,fc)

model = createpde() ;
g=triangle(X,area) ;
% Construit un triangle de cote X et d'aire area
geometryFromEdges(model,g); % geometryFromEdges for 2-D

%Conditions de bord :
%Les murs non chauffes sont a la temperature exterieure To = 10
applyBoundaryCondition(model,'dirichlet','Edge',[2,3],'u',10);

%Le mur chauffe est modelise par un flux rentrant
% on suppose que l on a mis un radiateur au niveau du mur
applyBoundaryCondition(model,'neumann','Edge',[1],'q',0,'g',50);

% Parametres de l'equation
a = 0;
c=1;
a=0;
f=fc;
% On choisit une maille adaptative, car les coins du triangle sont un probleme
[u,p,e,t] = adaptmesh(g,model,c,a,f,'maxt',5000,'par',1e-10);
pdeplot(p,e,t,'XYData',u,'ZData',u,'Mesh','off')
xlabel('x')
ylabel('y')
xlim([-X X])
ylim([-X X])
axis equal
result.u = u ;  % Valeur de u sur chaque mesh
result.p = p ;  % Coordonnees des points
result.t = t ;
result.e = e ;  % Reference de chacun des points
end
```

Une fois l'équation résolue, il faut maintenant s'interesser à calculer l'intégrale de cette fonction sur le triangle. Pour réaliser cela, nous avons programmé la fonction suivante, qui prend en paramètre la fonction uTriangle ci-dessus, et calcule l'intégrale approchée à l'aide des données fournies par l'algorithme précédent.

Listing 3: Calcul de l'integrale

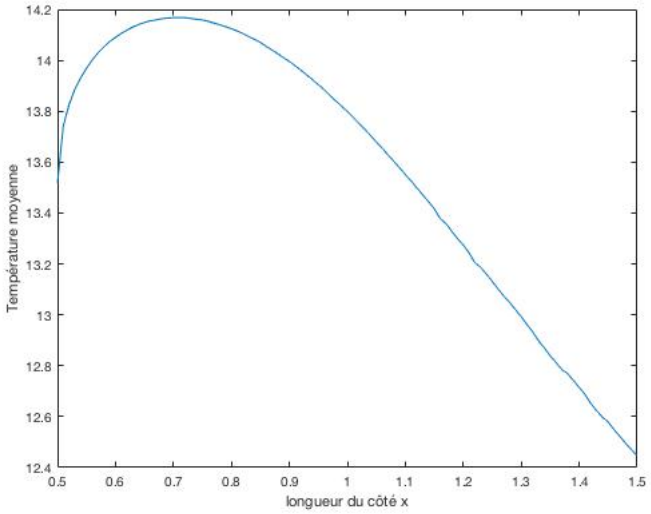
```
function I = Integrale(u)

coord = u.p ;     % Contient les coordonnees des sommets
indices = u.t ;   % Contient les references de chaque element
val = u.u ;

% On va calculer l integrale en evaluant la valeur de la fonction sur
% chaque petit triangle
I =0 ;
area = 0 ;
for i = 1:length(indices) ;           % Pour chaque triangle
    a = coord(:,indices(1,i)) ;       % Coord du 1er point
    b = coord(:,indices(2,i)) ;       % Coord du second point
    c = coord(:,indices(3,i)) ;       % Coord du troisieme point

    moy = (val(indices(1,i))+val(indices(1,i))+val(indices(1,i)))/3 ;
    area = area + 0.5*abs(a(1)*c(2)-a(1)*b(2)+b(1)*a(2)-b(1)*c(2)
    +c(1)*b(2)-c(1)*a(2)) ;
    I = I + moy*0.5*abs(a(1)*c(2)-a(1)*b(2)+b(1)*a(2)-b(1)*c(2)+c(1)*
    b(2)-c(1)*a(2)) ;
end
    I = I/area ;
end
```

On peut maintenant mettre en place l'algorithme pour determiner la solution de notre problème. On essaie donc différentes valeurs de x pour une aire fixée à 0.25.

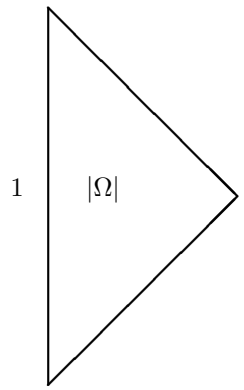
Listing 4: Script Principal

```
% Script principal
x = 0.5:0.01:1.5 ;
mat = [] ;
for i = 1:length(x) ;
        mat(i) = Integrale(uTriangle(x(i),1/4,0)) ;
end
plot(x,mat) ;
xlabel('longueur du cote x')
ylabel('Temperature moyenne')
```

Le programme nous retourne la courbe suivante :



On trouve alors un maximum aux alentours entre 0.70 et 0.71 (ce qui ressemble fortement à $\frac{\sqrt{2}}{2}$). Ce n'est pas vraiment une surprise, car on s'attendait à trouver une forme relativement régulière, et dans le cas présent, il s'agit du triangle rectangle équilatéral.



Procédons à quelques remarque. Tout d'abord, on observe que la figure "optimisée" est symétrique, ce qui semble s'accorder avec les conditions aux limites spatiales qui sont elles symétriques par rapport à l'axe des abscisses. Cela explique pourquoi dans la suite de l'étude, nous accorderons une place importante à la symétrie dans les figures recherchées.

Part III
Déplacement d'un point du polygone à aire constante

I - Un premier déplacement

L'idée général de l'algorithme repose sur le déplacement successif des différents points du polygone et cela en conservant l'aire totale du polygone. Pour cela on va déplacer un sommet sur la droite passant par les deux sommets voisins du sommet. Sur le schéma, nous avons représenté cette droite en rouge. L'idée derrière ce déplacement est d'exploiter la formule sur l'aire d'une triangle : $\mathcal{A} = \frac{\mathcal{B}.h}{2}$, ou $\mathcal{B}$ est la base du triangle et h est la hauteur du triangle. En déplacant les points de cette façon, nous pouvons conserver la hauteur et la base du triangle considéré par l'algorithme et par conséquent l'aire du triangle.

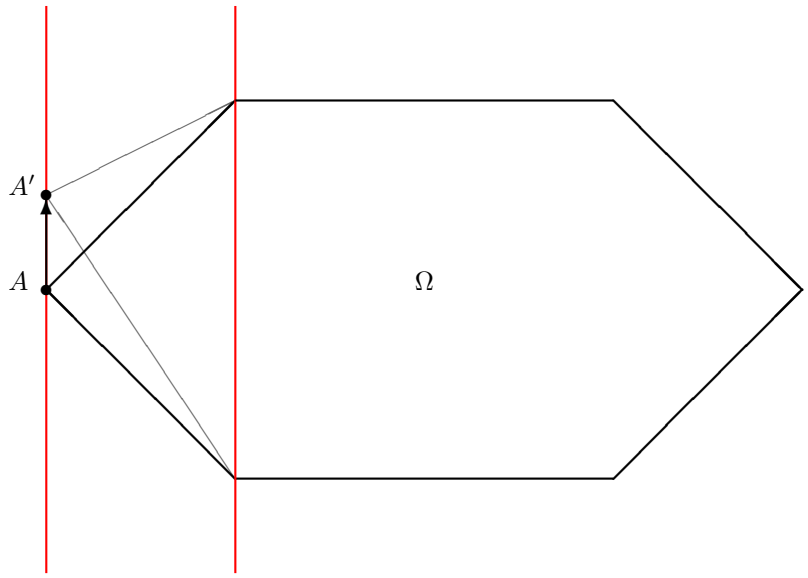


Figure : Déplacement du sommet A selon l'algorithme

L'idée est ensuite de faire varier les différents points du Polygone selon le même schéma de déplacement. Toutefois, la manière de choisir les points à déplacer et les critères d'arrêt sont encore à déterminer.

II - Un deuxième déplacement plus souple

L'algorithme présenté précédemment permet d'optimiser une polygone de façon relativement limitée, car les déplacements sont extrêmement spécifiques afin de conserver l'aire. Il peut alors être intéressant de chercher à trouver une méthode de déplacement des points qui permet une plus grande liberté dans le mouvement.

Contrairement à la méthode précédente, l'idée est ici de s'écarter de la notion de déplacement avec conservation de l'aire du polygone. Pour gagner en souplesse de déplacement l'idée va être d'essayer plusieurs déplacement élémentaires sur chacun des sommets amovibles pour les tester tous et trouver le meilleur déplacement. Dans l'algorithme précédente nous restions sur une droite afin de conserver l'aire du polygone, mais cette fois-ci nous allons tester des déplacements répartis sur un cercle de rayon fixé autour du point à déplacer (voir Fig.1)

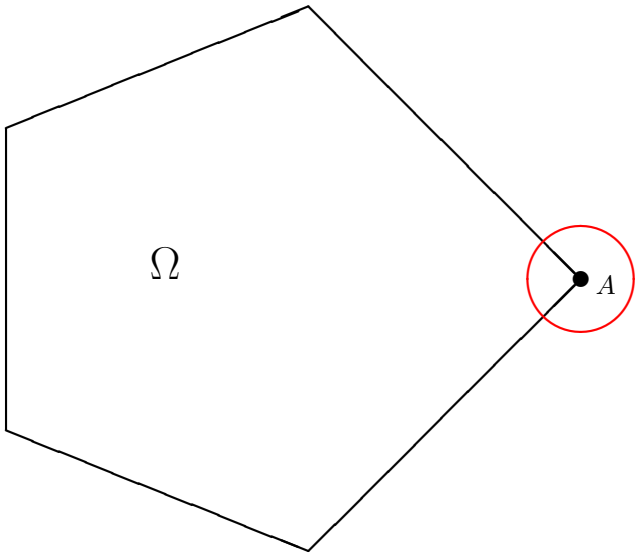


Fig.1 : Zone de déplacement du sommet A pour l'algorithme

Afin de discrétiser ce cercle, nous allons choisir un nombre de positions (ou directions) équitablement réparties sur le cercle. L'algorithme va ainsi pouvoir essayer toutes les configurations possibles sur le cercle afin de choisir le déplacement qui maximise la valeur moyenne.

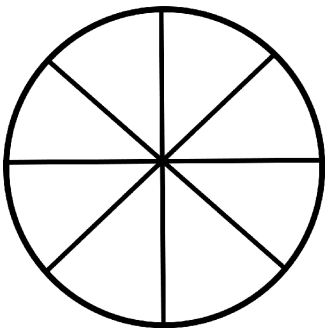


Fig.2 : Schéma dans une configuration à 8 positions

A chaque itération de l'algorithme, un déplacement va être effectué, cependant, les déplacements exposés ci-dessus ne permettent pas de conserver l'aire initiale du polygone. Nous avons donc pensé à combiner ce déplacement avec une contraction (ou décontraction) du polygone, afin qu'il possède la même aire que le polygone initial. Ainsi, nous pouvons garder la malléabilité au niveau du déplacement le tout en gardant une aire constante. La contraction du polygone est réalisée de façon assez naïve en déplaçant les points jusqu'à ce que l'on obtienne l'aire souhaitée (voir Fig.3). La contraction doit aussi obligatoirement se faire selon l'axe des abscisses car on doit absolument garder la longueur du mur modélisant le radiateur.

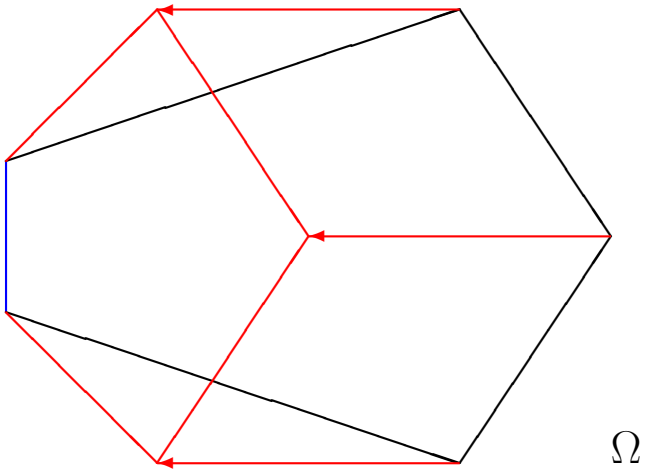


Fig.3 : Contraction selon l'algorithme

Regardons sur un exemple le résultat de l'algorithme

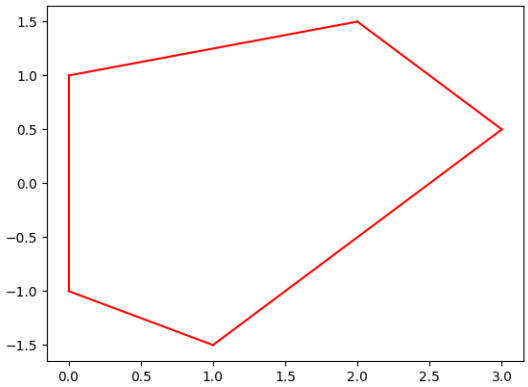


Figure : Polygone initiale

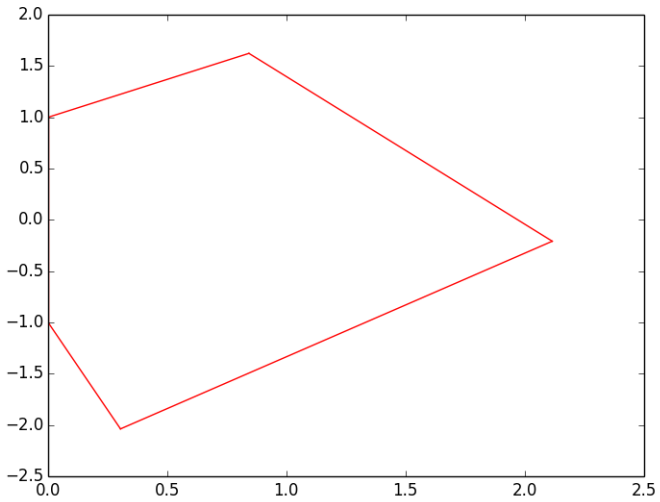


Figure : Polygone finale, après l'algorithme

Pour cet exemple comme pour les autres testés avec l'algorithme le résultat a toujours été analogue: étape après étape du processus le polygone se symétrise par rapport à l'axe de symétrie du problème (axe des abscisses, avec les valeurs choisies)

Pour les semaines à venir, nous allons essayer d'expliciter le profil des températures moyennes optimales obtenues selon les valeurs de pas utilisées en entrée de l'algorithme. Il est fort à parier qu'il soit possible de relier ce comportement à la "structure géométrique" des déplacements.

Voici un premier graphique obtenu, en raison d'un nombre d'itérations trop faibles utilisé il est difficilement interprétable.

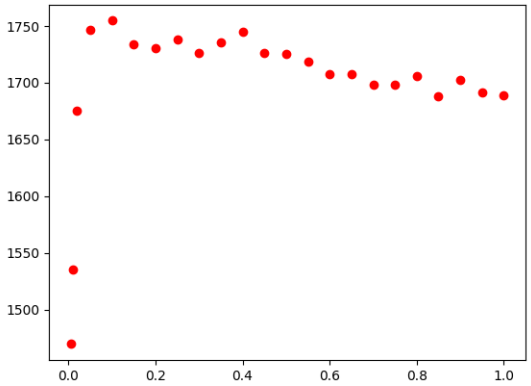


Figure: valeur de la température moyenne optimale obtenue en fonction du pas utilisé pour un polygone initial donné

Part IV
Implémentation des Algorithmes

I - Structure générale de l'algorithme

L'algorithme implémenté est un algorithme naïf qui se contente de bouger les différents points du polygone et de trouver le meilleur déplacement pour maximiser la température moyenne à chaque itération.

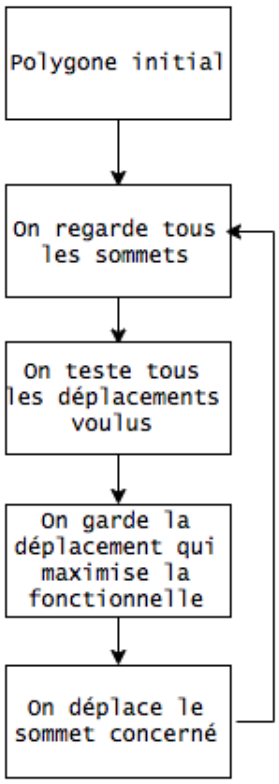


Fig 5.1 : Schéma simple de l'algorithme d'optimisation du polygone

II - Implémentation d'un algorithme sous Python

L'idée générale est de créer un objet sous python qui représente un polygone. Nous avons opté pour la création d'une classe appelée Polygon, qui est constituée d'une liste de points indépendants, qui peuvent être bougés au gré de nos envies.

Tout d'abord, nous disposons, via MATLAB, d'un outil puissant et efficace de résolution des équations différentielles: la PDE toolbox. Cependant, un inconvénient de cet outil est le temps d'exécution qui est élevé. Appelons *calcul_statio* le programme qui renvoie la valeur de la température moyenne de la chambre en régime stationnaire.

Présentons schématiquement les idées générales de l'algorithme **itératif** que nous avons choisi :

- L'algorithme prend en entrée les coordonnées du polygone initial dans le bon ordre ainsi que le pas p de déplacement;
- On entre dans la boucle: tous les sommets mobiles du polygones sont parcourus;
- Pour chaque sommet on teste les petits déplacements stables : pour chaque déplacement on applique *valueIntegral* au polygone
- On garde en mémoire le déplacement (éventuellement nul si pas d'amélioration) et la valeur de la température moyenne associée.
- On ressort de la boucle avec le meilleur déplacement et le sommet associé
- On applique le déplacement au polygone, puis on recommence (il s'agit bien d'un procédé itératif). Si plus aucun déplacement ne permet d'augmenter la valeur moyenne, on arrête l'algorithme

En termes de représentation informatique, nous avons décidé de modéliser les polygone par une classe *Polygon* sous Python, qui est représentée par une liste de points qui constituent les différents sommets. Ces différents points peuvent être modifiés grâce à des méthodes qui agissent sur cette instance de la classe *Polygon*.

III - Remarques et améliorations

Premièrement, nous pouvons remarquer que cet algorithme met en jeu une connexion entre python ou nous gérons les données relatives aux polygones et MATLAB pour faire les calculs. Cette connexion assez technique sur le plan purement informatique est couteuse en temps. Nous envisageons donc pour la suite de notre projet d'éventuellement trouver un module sur python pour y mener l'ensemble du processus.

Par ailleurs, la plus grande partie du temps d'exécution de l'algorithme a pour origine le maillage de la figure. Cette étape est essentielle pour la résolution d'une équation aux dérivées partielles. Pour optimiser notre algorithme, dans la mesure ou chaque étape n'engendre qu'une petite modification du polygone, nous pourrions éventuellement nous tourner vers des méthodes de remaillage efficaces.

Lorsque nous faisons des petits déplacements des sommets, il faut faire attention à ne pas casser "la convexité" de la figure, ce qui pourrait passer aussi par un cas singulier où notre polygone aurait un coté en moins. Nous allons ajouter un test à notre algorithme pour nous assurer qu'il n'y est pas de problème à cause de ces cas limites.

Ensuite, pour améliorer la complexité de l'algorithme, nous envisageons d'adapter la méthode du gradient et de chercher à identifier des directions privilégies.

Ensuite, une grande limitation de cet algorithme est que la valeur du pas influe grandement sur le résultat final. Contrairement à ce que nous pourrions penser naivement un pas petit n'améliore pas nécessairement la qualité du polygone finale (on peut rester bloquer sur un maximum local).

A l'heure actuelle, nous n'avons pas pris en compte les cas difficiles, et nous préférons partir de formes polygonales relativement concaves afin d'arriver à la solution sans que les déplacement ne rendent le polygone convexe. Dans certains cas, le polygone va "croiser" ses arêtes, et générer une erreur.

Les figures polygonales très dendritiques sont aussi problématiques, car la résolution de l'équation différentielle sur des zones avec un très faible angle explosent et donnent des résultats aberrants.

Part V
Résultats des algorithmes

I - A quoi nous attendons nous ?

A première vue il est assez difficile de prévoir la forme des polygones optimaux même si on peut éventuellement "intuiter" le résultat en espérant trouver des figures régulières (polygones réguliers). Pour se faire une vague idée des résultats, nous avons réalisé une sorte de classification des polygones, en générant une vingtaine de polygones de même aire et de les associer à une couleur proportionnelle à la valeur de leur température moyenne. Nous obtenons ainsi les résultats suivants :

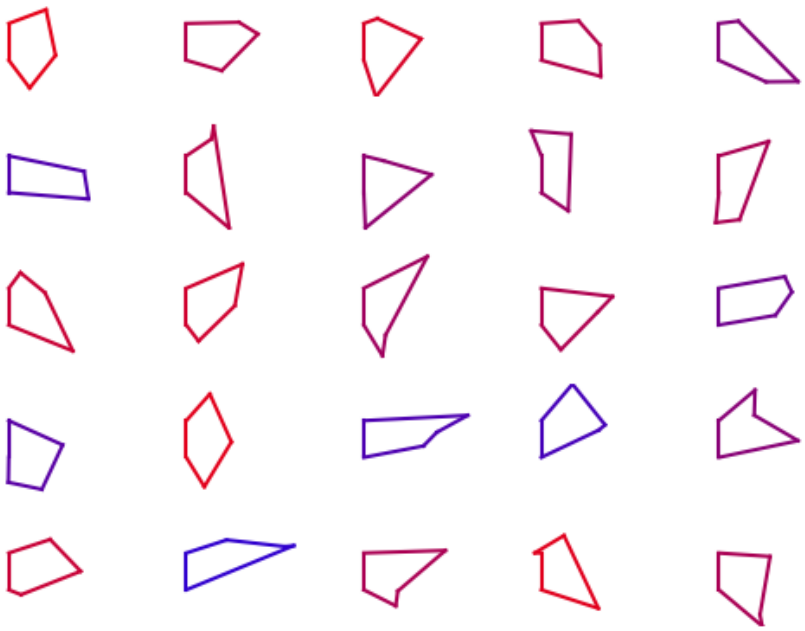


Fig : Carte de polygones de même aire (Les polygones avec une haute valeur moyenne sont coloriés en rouge et en bleu si elle est basse)

De cette carte, on peut vaguement tirer quelques informations. Par exemple, il semble que les polygones "massés" autour du mur chauffé semblent posséder une meilleure valeur de température moyenne. De la même façon les figures longues, fines, assez éloignées du mur chauffé semblent posséder une valeur moyenne plus basse. Pour synthétiser cette vision, on pourrait dire que les figures les plus optimisées sont celles qui diminuent au maximum la distance de chacun des points intérieur au mur chauffé. Il semblerait aussi avec ce raisonnement que les figures symétriques soient plus optimisées.

II - Cas des déplacements sur les droites parallèles

Moralement on s'attend lors d'une optimisation à ce que les figures deviennent plus symétriques et tassées proche du mur. C'est effectivement ce que l'on distingue lors des simulations avec la première méthode de déplacement. Toutefois cela n'est pas frappant à cause du faible nombre d'itérations.

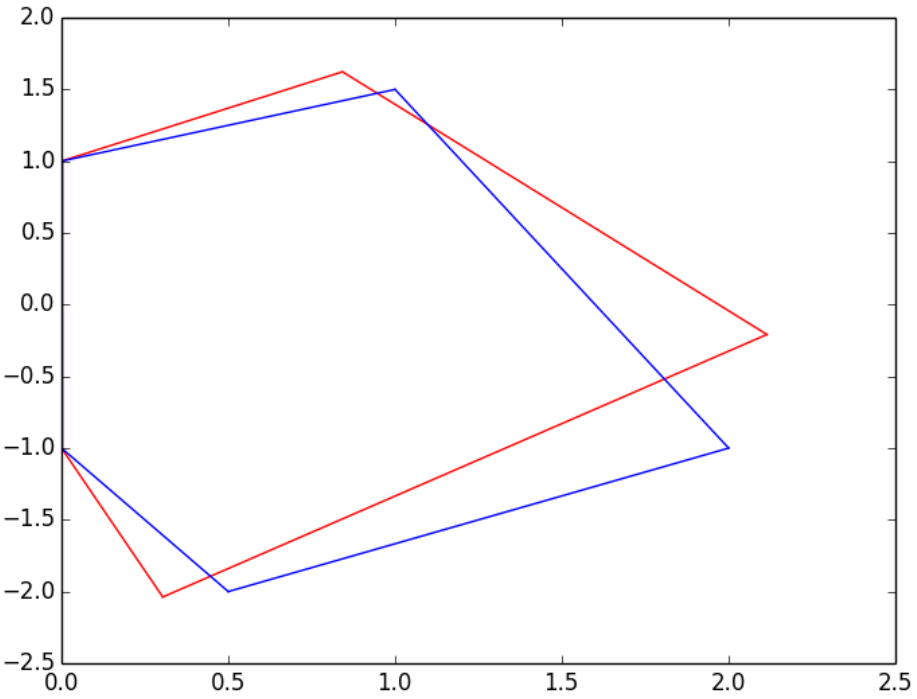


Fig 5.1 : Optimisation d'un polygone avec la première méthode pour 50 itérations (En rouge le polygone optimisé, et en bleu le polygone initial)

On constate bien une amélioration globale de la température moyenne après l'optimisation, cependant ce n'est pas très satisfaisant pour deux raisons majeures :

- L'algorithme se retrouve très vite bloqué par sa faible liberté de mouvement
- La forme optimisée est trop dépendante de la figure initiale choisie, et dépend énormément du pas de déplacement choisi

Ce sont en partie ces constats qui nous ont poussé à chercher d'autres méthodes de déplacement des points, afin de gagner une plus grande liberté en terme de déplacement.

III - Cas des déplacements améliorés

La plus grande liberté de mouvement donnée par la méthode utilisée nous laisse espérer que les solutions données par cet algorithme seront meilleures que celles données par l'algorithme précédent. Afin de constater cela, nous allons réaliser une comparaison sur deux exemples de polygone différents, mais de même aire pour 50 itérations :

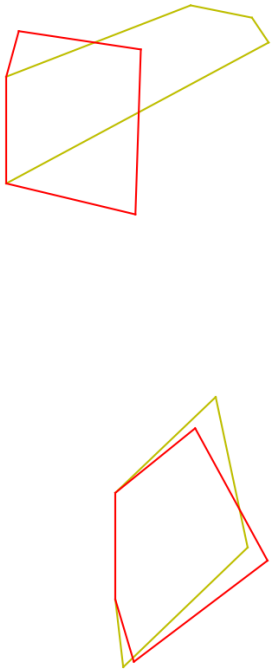


Fig : En jaune le polygone initial, et en rouge le polygone optimisé (voir Annexe pour les détails)

La conjecture émise dans la première section semble bien s'appliquer au vue de la première optimisation présentée ci-dessus. La deuxième est effectivement moins flagrante.

Il est cependant assez difficile de comparer les deux méthodes sérieusement à partir de quelques observations. Nous avons donc réalisé quelques comparaisons sur des polygones, mais étant donné le temps de simulation, nous avons du nous contenter d'une dizaine de comparaisons.

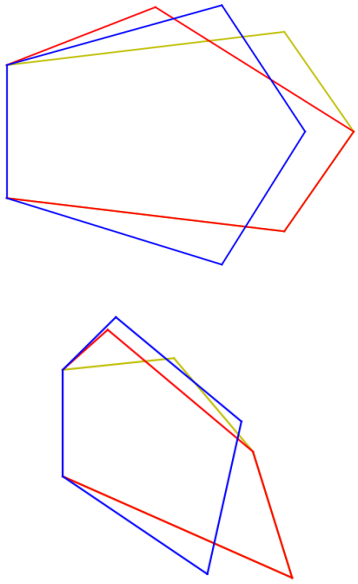


Fig : Simulation sur des polygones convexes générés aléatoirement. En jaune le polygone initial, et en rouge le polygone optimisé avec la méthode 1, et en bleu avec la méthode 2 pour 20 itérations

On constate clairement une nette avance de la seconde méthode. La méthode 1 étant très limitée en terme de liberté de mouvement, elle n'a pas permis d'optimiser correctement les figures montrée dans la simulation ci-dessus. Dans les simulations ci-dessus, les polygones sont générés aléatoirement et sont peu réguliers, ce qui tend à discriminer encore plus la méthode 1, car sa trop faible liberté de mouvement la contraint extrêmement rapidement, et elle ne trouve plus de déplacements optimisés dés quelques itérations (L'algorithme de la méthode 1 s'arrête au bout de 10 itérations dans la deuxième figure par exemple)

Cet algorithme semble bien plus satisfaisant que le premier dans la mesure ou pour un même nombre d'itérations la figure est largement plus optimisée.

IV - Exploitation des symétries

Comme l'on s'attend plutôt a ce que les figures optimales soient symétriques, il peut être intéressant de partir de figures déjà symétriques et d'exploiter la symétrie du polygone pour le déplacer avec la méthode de nôtre choix, mais en conservant la symétrie. Les avantages sont multiples si l'on tient compte de cela. Tout d'abord, on réduit par 2 le nombre de calculs de température moyenne ce qui diminue drastiquement le temps de simulation. D'autre part, il y a une plus rapide convergence de l'algorithme vers un polygone optimal lorsque l'on tient compte des symétrie.

En reprenant le polygone de l'exemple précédent, et en appliquant une fois l'algorithme basique et une fois en exploitant les symétries avec les mêmes paramètres, on obtient les résultat suivants :

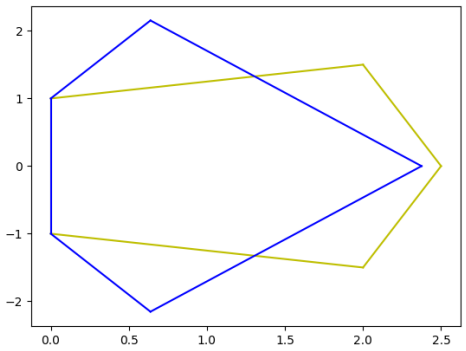


Fig : Exploitation des symétries (18 itérations, Valeur : 1763)

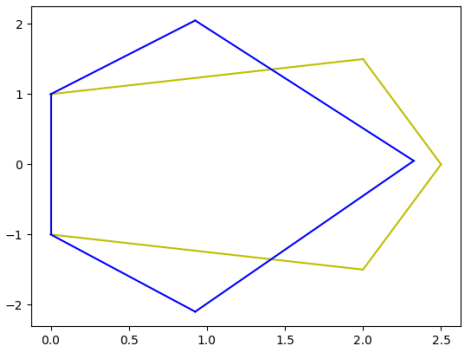


Fig : Non Exploitation des symétries (50 itérations, Valeur : 1733)

Pour trouver la véritable figure optimale il peut être judicieux de plutôt exploiter des figures symétriques avec cet algorithme, car la solution optimale a de forts risques d'être symétrique. De même on a très envie de s'approcher d'une figure régulière, et il peut être intéressant de tester l'algorithme avec une forme initiale très proche d'une forme régulière et un pas assez fin, afin de voir s'il converge vers cette forme ou non.

Annexes

Classe Vector

Cette première classe permet juste de représenter des vecteurs dynamiques sous Python.

Listing 5: Création de la classe Vecteur

```
class Vector :
    """ Represente un vecteur """

    def __init__(self,x,y) :
        self.x = x
        self.y = y

    def __str__(self) :
        return ("v(%f,%f)" % ( self.x, self.y))


    # Normalise le vecteur
    def normalize(self) :
        norm = (self.x ** 2 + self.y ** 2) ** 0.5
        self.x = self.x / norm
        self.y = self.y / norm
```

Classe Vertex

La classe Vertex permet de représenter des sommets mobiles sous Python. Chaque polygone est ainsi constitué de plusieurs objets de classe Vertex (de plusieurs sommets). La classe contient notamment une fonction *move* permettant de déplacer un point selon une droite.

Listing 6: Création d'une classe Vertex

```
class Vertex :
    """
     La classe Vertex est une classe qui represente les coordonnees des points
     d'un polygone donne
    """

    def __init__(self, x, y) :

        self.x = x
        self.y = y

    def __str__(self) :
        return ("(%f,%f)" % (self.x, self.y) )

    def __copy__(self, vertex) :
        self.x = vertex.x
        self.y = vertex.y

    #-----------------------------------------------------------------------
    # Deplace le sommet de dx selon x et dy selon y

    def update(self, dx, dy) :
        self.x += dx
        self.y += dy


    #-----------------------------------------------------------------------
    # deplace le point dans la direction du vecteur unitaire vector

    def move(self, vector, dl) :
        # On deplace le point selon le vecteur
```

```
        dx = dl * vector.x
        dy = dl * vector.y
        self.update(dx, dy)
```

Classe Polygon

La classe Polygon est l'élément central de notre optimisation de forme. En effet nous cherchons à optimiser des objets de forme polygonale.
Chaque objet Polygon est constitué d'objets de classe Vertex qui contiennent les coordonnées des sommets. Ces Vertex sont contenus dans un attribut de type liste, et ils sont ordonnés dans le sens horaire. On commence conventionnellement par mettre les sommets liant le côté traversé par le flux. C'est très important, car le calcul de la température moyenne sous Matlab dépend de l'ordre des côtés, car il faut choisir des conditions aux limites différentes sur chacun des côtés.
Les méthodes de cette classe Polygon sont les suivantes :

1. *init* qui initialise la classe avec une liste **ordonnée** de sommets de type Vertex
2. *deepCopy*() qui permet de copier un objet
3. $getx(i)$ et $gety(i)$ qui retournent les coordonnées x et y du sommet numéroté i
4. $directorVertice(i)$ qui retourne le coefficient directeur de la droite passant par le sommet i, et parallèle à ses deux voisins les plus proches (voir la droite rouge sur le schéma du déplacement)
5. $directorSide(i)$ qui retourne le coefficient directeur de la droite passant par le côté numéroté i
6. *buildGeometry*() qui construit une liste représentant le polygone en Matlab. Cette liste est exportée sous Matlab pour être traitée.
7. $move(i, dl)$ qui bouge le sommet numéroté i d'une distance dl (Attention : dl peut être négatif)
8. $valueIntegral(i, dl, eng)$ qui calcule la température moyenne dans le polygone lorsque le sommet i est déplacé d'une longueur dl. Cette fonction est déclinée en $valueIntegralOS(i, dl, eng)$ (OS:odd sides and symetrical) qui prend en compte la symétrie dans le mouvement
9. $plotPY(color)$ qui trace sur le canvas le polygon (pour faire des représentations graphiques
10. *area*() qui calcule l'aire du polygone grâce à un module spécifique
11. *degSymetrie*() qui calcule le "degré de symétrie" du polygone grâce à un module encore une fois

Listing 7: Création de la classe Polygon

```
class Polygon :

    """
        La classe Polygon contient des Vertex (Sommets). L'ensemble de ces sommets
        forme un polygon

        Les attributs de cette classe sont :
           N : le nombre de côtés
           vertices : la liste contenant les sommets

    """

    def __init__(self, *args) :
        # Nombre de côtés dans le polygone
        self.N = len(args)

        # Liste contenant les sommets (vertex)
        self.vertices = []
        for vertex in args :
            self.vertices.append(vertex)

    def deepCopy(self):
        copy = Polygon()
        copy.N = self.N
        for vertex in self.vertices :
            copy.vertices.append(vertex.deepCopy())
        return copy


    # Print
    def __str__(self) :
        res = "("
        for vertex in self.vertices :
            res = res + vertex.__str__() + ","
        return res[:-1]+')'


    #--------------------------------------------------------------------
    # Retourne les coordonnées x et y d'un sommet
    #
    # Retourne la coordonnée x du sommet i
    def getx(self, i) :
        return self.vertices[i % self.N].x

    # Retourne la coordonnée y du sommet i
    def gety(self, i) :
        return self.vertices[i % self.N].y



    #--------------------------------------------------------------------
    # La fonction findCoef renvoie le coefficient directeur de la droite passant
    # par les sommets i-1 et i+1

    def directorVertice(self, i) :
        dx = self.getx(i + 1) - self.getx(i - 1)
        dy = self.gety(i + 1) - self.gety(i - 1)
        res = Vector(dx, dy)
        res.normalize()
        return res



    #--------------------------------------------------------------------
    # calcule le vecteur directeur de la droite passant par le côté i
    def directorSide(self, i) :
```

```
        dy = self.gety(i + 1) - self.gety(i)
        dx = self.getx(i + 1) - self.getx(i)
        res = Vector(dx, dy)
        res.normalize()
        return res


    #-----------------------------------------------------------------
    # Cette méthode construit une géométrie prête à l'exportation sous matlab

    def buildGeometry(self) :

        #    On initialise une liste avec l'argument 2, qui est le
        #    code correspondant a une forme polygonale sous
        #    matlab, et self._N est le nombre de côtés
                #    Cette fonction renvoie une matrice prête a être
                #    employée dans la fonction Matlab decsg(mat)

        mat = [[2], [self.N]]
        for vertex in self.vertices :
            mat.append([vertex.x])
        for vertex in self.vertices :
            mat.append([vertex.y])
        return mat

    #-----------------------------------------------------------------
    #
    #    Déplacement d'un point du polygone selon l'algorithme
    #    i correspond au numéro du sommet et dl à la longueur du déplacement

    def move(self, i, dl) :
        vector = self.directorVertice(i)
        self.vertices[i % self.N].move(vector, dl)


    #-----------------------------------------------------------------
    #
    #    Calcul de la valeur de l'intégrale pour un déplacement
    #    du sommet i d'une longueur dl. La méthode ne modifie ainsi
    #    pas le polygone lorsque elle est executée

    def valueIntegral(self, i, dl, eng) :
        self.move(i, dl)
        mat = matlab.double(self.buildGeometry())
        value = eng.computeIntegral(mat)
        self.move(i, -dl)
        return value

    #-----------------------------------------------------------------
    #
    #   La fontion calcul calcul aussi l'intégrale, mais pour un
    #   un déplacement symétrique du polygone symétrique à côtés
    #   impairs (On exploite donc la symétrie)

    def valueIntegralOS(self, i, dl, eng) :
        self.move(i, dl)
        self.move(self.N - i, -dl)
        mat = matlab.double(self.buildGeometry())
        value = eng.computeIntegral(mat)
        self.move(i, -dl)
        self.move(self.N - i + 1, dl)
        return value


    #-----------------------------------------------------------------
    # Fonction de traçage

    def plotPY(self,color) :
```

```
        for k in range(self.N) :
            plt.plot([self.vertices[ k % self.N].x,
                      self.vertices[(k + 1) % self.N].x],
                     [self.vertices[k % self.N].y,
                      self.vertices[(k + 1) % self.N].y],
                     color
                     )

        #plt.show()


    #-----------------------------------------------------------------------
    # Retourne l'air du polygone

    def area(self) :

        poly = [[self.getx(i), self.gety(i)] for i in range(self.N)]
        return aire_poly(poly)

    #-----------------------------------------------------------------------
    # Retourne l'air du polygone

    def degSymetrie(self, step) :

        poly = [[self.getx(i), self.gety(i)] for i in range(self.N)]
        return deg_sym(poly, step)
```

Calcul de la valeur moyenne sous Matlab

Le script suivant est une fonction prenant en argument une forme géométrique sur laquelle va être résolue l'EDP. Voici le fonctionnement de la fonction résumée :

1. La géométrie importée est implantée dans l'objet model qui va servir à résoudre l'équation differentielle
2. avec *applyBoundaryCondition* on applique les conditions aux limites sur le modèle étudié (ici dirichlet sur tous les côtés sauf le premier avec une valeur 10, et newmann sur le premier côté avec une valeur 10000
3. Ensuite on résout l'équation de la chaleur (f=0 car pas de termes de source). Les variables u, p, t contiennent respectivement les valeurs scalaires de la fonction, les coordonnées des sommets, et les références de chaque sommets
4. A l'aide de ces dernières variables, on calcule l'intégrale de la fonction solution (C'est à dire la température moyenne)

Listing 8: Script Matlab pour le calcul de la valeur moyenne sur une géométrie donnée

```
function I = computeIntegral(mat)


model = createpde() ;
g=decsg(mat);
geometryFromEdges(model,g); % geometryFromEdges for 2-D


edges = [2:1:(size(mat)-1)/2];
%Conditions de bord :
%Les murs non chauffés sont  la température éxterieure To = 10C
applyBoundaryCondition(model,'dirichlet','Edge',edges,'u',10);
```

```
%Le mur chauffé est modelisé par un flux rentrant , on suppose que l'on a
%mis un radiateur au niveau du mur
applyBoundaryCondition(model,'neumann','Edge',[1],'q',0,'g',10000);

a = 0;
c=1;
a=0;
f=0;
[u,p,e,t] = adaptmesh(g,model,c,a,f,'Par',0.1,'tripick','pdeadworst',
                      'MesherVersion','R2013a');



%Le résultat est ainsi renvoyé
%On calcule l'intégrale de la fonction renvoyée

coord = p ; % Contient les coordonnées des sommets
indices = t ; % Contient les références de chaque élément
val = u ;

% On va calculer l'intégrale en évaluant la valeur de la fonction sur
% chaque petit triangle

I =0 ;
area = 0 ;
for i = 1:length(indices) ; % Pour chaque triangle
    a = coord(:,indices(1,i)) ;      % Coord du 1er point
    b = coord(:,indices(2,i)) ;      % Coord du second point
    c = coord(:,indices(3,i)) ;      % Coorddu troisième point

    moy = (val(indices(1,i))+val(indices(1,i))+val(indices(1,i)))/3 ;
    area = area + 0.5 * abs( a(1) * c(2 )- a(1) * b(2) + b(1) * a(2) - b(1) *
    c(2) + c(1) * b(2) - c(1) * a(2)) ;
    I = I + moy * 0.5 * abs( a(1) * c(2) - a(1) * b(2) + b(1) * a(2) - b(1) *
    c(2) + c(1) * b(2) - c(1) * a(2)) ;
end
    I = I / area ;
```

Implementation de l'algorithme d'optimisation sous Python

On commence tout d'abord par créer une fonction *bestValue* qui prend en argument un polygone, la valeur de l'intégrale sur ce polygone (pour un soucis d'optimisation de temps de calcul), le numéro du sommet que l'on veut étudier (i), le nombre de test que l'on veut effectuer de part et d'autre du sommet (nbTest), la taille du déplacement (dl) et le moteur matlab (eng)
Cette fonction renvoie la valeur maximal, et le déplacement associé à cette valeur maximale dans une liste. Cette liste est de la forme [Valeur Maximale, déplacement associé (scalaire algébrique)]

Listing 9: Fonction permettant de trouver la meilleure valeur de la temperature pour le déplacement d'un seul sommet

```
# ================================================================================
#                                Best Value of a vertex
# ================================================================================

#   On cherche la position du sommet i qui maximise la fonctionnelle de forme
#   la fonction prend en paramètres :
#
#   - initValue : la valeur de l'intégrale du polygone initial (pour optimiser
#     et ne pas avoir à le calculer à chaque fois)
#
#   - dl : la longueur du pas
#
#   - i : le numéro du sommet
#
#   - nbTest : le nombre de valeur testée de part et d'autre du sommet, en tout
#     2*nbTest valeurs sont testées sur chaque sommet
#
#   - eng : le moteur matlab
#

def bestValue(polygon, initValue, i, nbTest, dl, eng) :

    #   On stocke les valeurs des intégrales liées à chacun des déplacements
    #   dans une liste
    #   right correspond aux déplacements à droite du point
    #   left correspond aux déplacement à gauche du point

    left = [polygon.valueIntegral(i, -j * dl, eng) for j in range(nbTest)]
    right = [polygon.valueIntegral(i, j * dl, eng) for j in range(nbTest)]
    L = np.array(left + [initValue] + right)

    #   On cherche le maximum dans cette liste, puis on trouve l'index de ce
    #   maximum, qu'on appelle indexMax
    indexMax = np.argmax(L)

    #   Le résulat retourné contient la valeur maximum de l'intégrale,
    #   et le déplacement associé à celui-ci
    return [L[indexMax], (indexMax - nbTest) * dl]
```

Désormais, on peut écrire la boucle principale de l'algorithme. Comme il s'agit de la première version, nous l'appelons *naiveMainloop* car elle n'est pas très optimisée.
Les arguments pris en compte sont :

1. La forme initiale polygon
2. la taille du pas de déplacement dl
3. le nombre de tests (nbTest) à effectuer sur chaque déplacement
4. le nombre d'itérations maximal (nbIteration)
5. une liste values pour stocker les valeurs de la température moyenne lors de chaque itération
6. l'instance de matlab en cours (eng)

Le principe de fonctionnement est tel qu'il est énoncé dans la description initiale de l'algorithme. On parcourt tous les sommets, et on retient seulement un seul déplacement de sommet qui maximise la température moyenne.

Listing 10: Boucle Principale

```
# ============================================================================
#                                Naive Mainloop
# ============================================================================

#   Cette boucle naive se contente de parcourir les sommets pour trouver le
#   meilleur déplacement possible


def naiveMainloop(polygon, dl, nbTest, nbIteration, values, eng) :
    #   Conservation des données de la temperature moyenne pour chaque itération
    #   values est une liste vide destinée à conserver les valeurs de l'intégrale

    if nbIteration == 0 :
        print("Fin de la simulation")
        return 0

    initValue = polygon.valueIntegral(0,0,eng)

    #   On cherche le petit déplacement qui maximise notre fonctionnelle
    #   de forme lors d'une itération de l'algorithme
    #   On examine chacun des sommets indépendamment

    max = [0,0]                             # Initialisation du maximum
    rank = 0
    for i in range(2, polygon.N) :
        val = bestValue(polygon, initValue, i, nbTest, dl, eng)
        if val[0] > max[0] :
            max = val
            rank = i

    #   Si la valeur maximum est atteinte pour un déplacement nul,
    #   on arrête la simulation
    if max[1] == 0 :
        print("Il reste " + str(nbIteration) + " itérations")
        return 0

    # Sinon on bouge un sommet
    polygon.move(rank, max[1])
    values.append(max[0])

    # Appel récursive de la fonction
    naiveMainloop(polygon, dl, nbTest, nbIteration - 1, values, eng)
```

Modules complémentaires