

## Projet 2A : La chambre la moins froide

March 8, 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 vis-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  $\Omega$  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  $\Omega$  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 égalisées 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 dès 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  $\Phi$  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

# Etude préliminaire: le cas des triangles

## I - Cadre général de l'étude

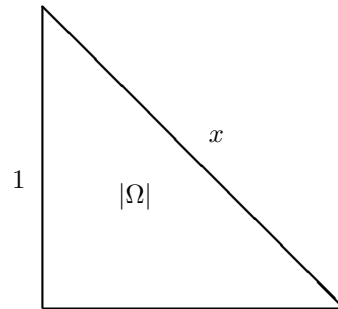
On va considérer le problème simple d'une pièce triangulaire dans toute cette partie. On considère aussi une application  $T_S$  qui a une valeur  $x$  va associer un triangle d'aire  $S$  et de côtés de longueur  $x$  et 1. La fonction  $u_\Omega$  correspond à la solution de l'équation précédente sur le domaine  $\Omega$ , qui est dans le cas ici présent un triangle. On va noter  $M$  la fonction qui renvoie la moyenne de  $u_\Omega$  sur  $\Omega$  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  $\mathbb{R}^2$

$$T_\Omega : \mathbb{R} \mapsto E$$

$$x \mapsto \text{Triangle de côté 1 et } x, \text{ d'aire } |\Omega|$$



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 connaît 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  $\Omega$  et de la longueur du côté  $x$ . Cet objet geometry définit la forme de la zone sur laquelle on va résoudre l'équation différentielle.

Listing 1: Construction d'une géométrie Triangulaire

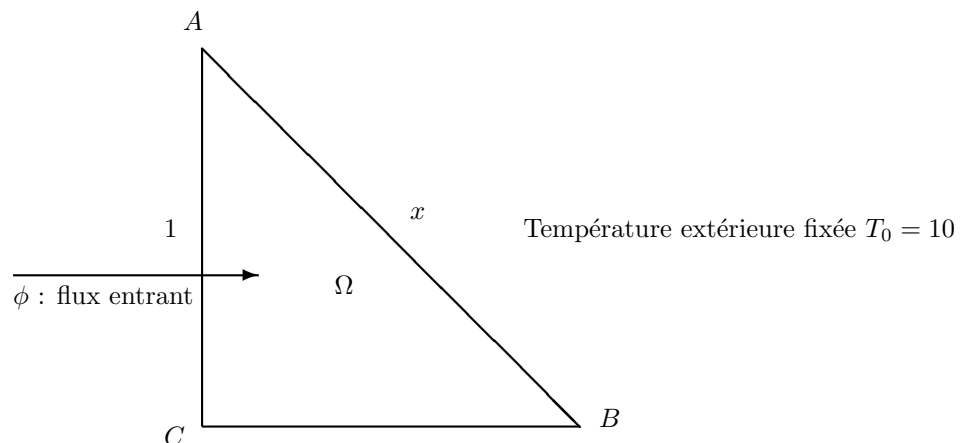
```
function g = triangle(X,area)

% 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

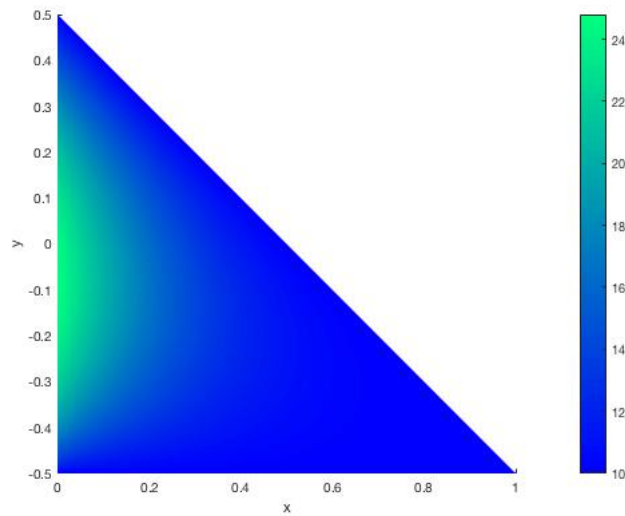
On va considérer ici des conditions relativement simples. Premièrement, on se place en régime stationnaire, c'est à dire que le terme  $\frac{\partial u}{\partial t}$  est nul. De plus, on fixe une fonction de chauffage nulle. Enfin, les conditions aux limites sont fixées telles que le côté de longueur 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 :

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

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'intéresser 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

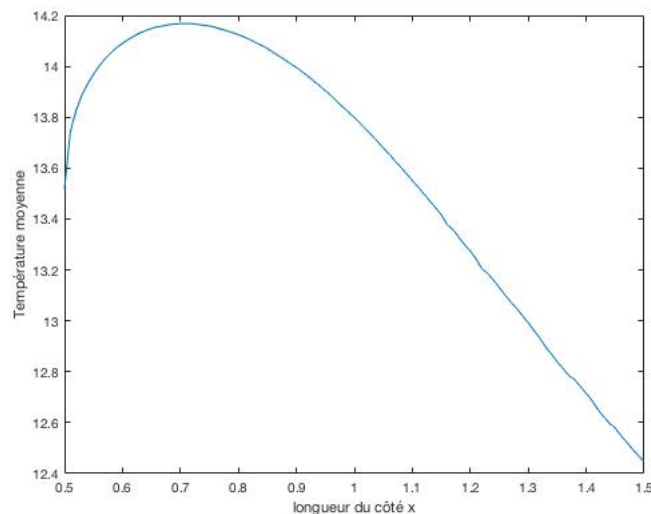
    moy = (val(indices(1,i))+val(indices(2,i))+val(indices(3,i)))/3 ;
    area = area + 0.5*abs(a(1)*c(2)-a(1)*b(2)+b(1)*a(2)-b(1)*c(2)
    +c(1)*b(2)-c(1)*a(2)) ;
    I = I + moy*0.5*abs(a(1)*c(2)-a(1)*b(2)+b(1)*a(2)-b(1)*c(2)+c(1)*
    b(2)-c(1)*a(2)) ;
end
    I = I/area ;
end
```

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.

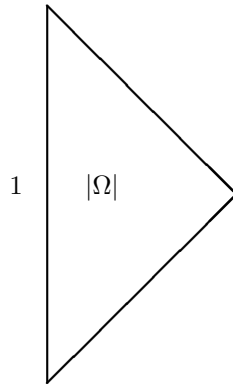
Listing 4: Script Principal

```
% 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

# Implémentation d'un premier algorithme sur Python

## I - Déplacement d'un sommet du polygone avec aire constante

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'un triangle :  $\mathcal{A} = \frac{B \cdot h}{2}$ , où  $B$  est la base du triangle et  $h$  est la hauteur du triangle. En déplaçant 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.

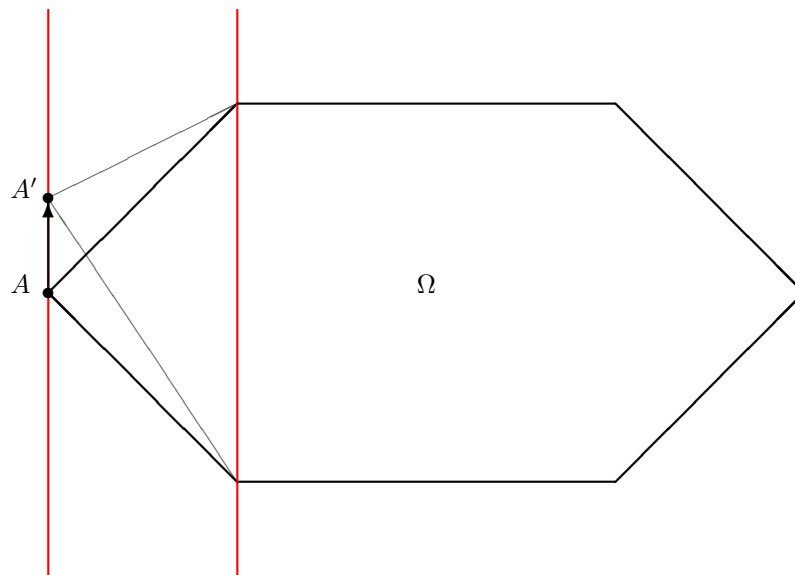


Figure : Déplacement du sommet  $A$  selon l'algorithme

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 - Implémentation d'un algorithme sous Python

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

Présentons succinctement les idées générales de l'algorithme que nous avons choisi :

- On parcourt les sommets du polygone, puis on teste deux petits déplacement de chacun des sommets
- On garde en mémoire le déplacement et la valeur de l'intégrale associé à chacun des petits déplacements
- Une fois que tous les sommets ont été parcourus, on choisit le meilleur déplacement par rapport à tous les sommets
- On applique le déplacement au polygone, puis on recommence

Le calcul de l'intégrale sur le polygone se fait à l'aide de Matlab, qui permet de résoudre directement l'équation différentielle sur le polygone. On a juste à importer la géométrie sous Matlab, puis on utilise la PDE toolbox pour résoudre l'équation différentielle partielle sur notre figure. Le seul défaut de cette méthode, est son coût en remaillage, qui prend pas mal de temps. Le temps de calcul de cet algorithme est ainsi assez élevé.

### III - Résultats de cet algorithme

## Annexes

Listing 5: Création de la classe Vecteur

```
class Vector :
    """ Représente 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
```

Listing 6: Création d'une classe Vertex

```
class Vertex :
    """
        La classe Vertex est une classe qui représente les coordonnées 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

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

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

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

    def move(self, vector, dl) :
        # On déplace le point selon le vecteur
        dx = dl * vector.x
        dy = dl * vector.y
        self.update(dx, dy)
```

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 cotes
        _vertices : la liste contenant les sommets
    """

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

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

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

#-----
# Retourne les coordonnees x et y d'un sommet
#
# Retourne la coordonnee x du sommet i
def getx(self, i) :
    return self.vertices[i % self.N].x
# Retourne la coordonnee 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 cote 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 methode construit une geometrie prete a 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 cotes. Cette fonction renvoie une matrice prete a etre
# employee 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

#-----
# Deplacement d'un point du polygone selon l'algorithme
# i correspond au numero du sommet et dl a la longueur du
# deplacement

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

#-----
# Calcul de la valeur de l'integrale pour un deplacement du sommet i
# d'une longueur dl
# La methode ne modifie ainsi pas le polygone lorsque elle est executee

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

#-----
# Fonction de tracage

def plotPY(self) :

    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],
                  '-r'
                  )

    plt.show()

```

Listing 8: Trouver le meilleur déplacement

```

# On cherche la position du sommet i qui maximise
# la fonctionnelle de forme

def bestValue(polygon, i, dl, eng) :
    # On initialise matlab, puis calculons la valeur de l'integrale initiale

    L = np.array([polygon.valueIntegral(i, -dl, eng),
                  # Valeur de l'integrale pour le deplacement a gauche
                  polygon.valueIntegral(i, 0, eng),
                  # Valeur de l'integrale sans deplacement
                  polygon.valueIntegral(i, dl, eng)])
                  # Valeur de l'integrale pour le deplacement a droite

    index = np.argmax(L)
    # Retourne l'index du maximum de cette liste
    return [L[index], (index - 1) * dl]

```

Listing 9: Boucle principale de l'algorithme

```
def oneIteration(polygon, dl, eng) :  
  
    # On cherche le petit déplacement qui maximise notre fonctionnelle  
    # de forme lors d'une iteration de l'algorithme  
    # On examine chacun des sommets independamment  
    max = [0,0]  
    rank = 0  
    for i in range(2, polygon.N) :  
        val = bestValue(polygon, i, dl, eng)  
        print("val_="+str(val))  
        if val[0] > max[0] :  
            max = val  
            rank = i  
    polygon.move(rank, max[1])
```