Équation de Poisson : résolution numérique

1. Introduction

L'équation de Poisson à deux dimensions est :

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = s(x,y) \tag{1}$$

où u(x,y) est la fonction inconnue et s(x,y) la fonction source, éventuellement nulle (équation de Laplace).

Un exemple d'équation de Poisson est celle vérifiée par le potentiel électrostatique :

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = -\frac{\rho}{\epsilon} \tag{2}$$

où ρ est la densité volumique de charge électrique et ϵ la permittivité électrique du milieu.

Un autre exemple est l'équation de la chaleur en régime stationnaire vérifiée par la température :

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = -\frac{\sigma}{\lambda} \tag{3}$$

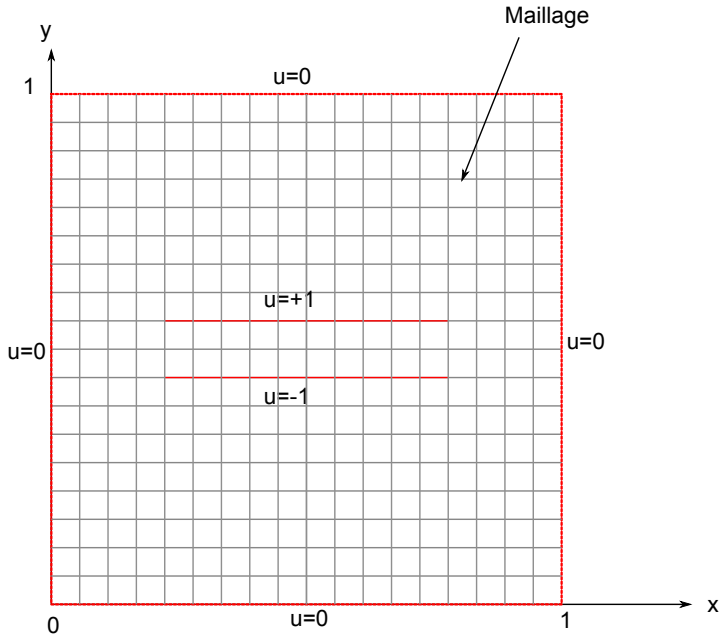
où $\sigma(x,y)$ est la source thermique et λ la conductivité thermique.

En mécanique des fluides, l'écoulement bidimentionnel d'un fluide parfait (c.a.d. non visqueux) et irrotationnel autour d'un obstacle peut être décrit par un potentiel de vitesse Φ tel que : $\overrightarrow{v} = \overrightarrow{grad}(\Phi)$. Si l'écoulement est de plus incompressible, la divergence de la vitesse est nulle et on obtient l'équation de Laplace :

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0 \tag{4}$$

La condition limite sur un obstacle consiste à écrire que la vitesse du fluide est tangente à la surface de l'obstacle en tout point de celui-ci, c'est-à-dire que la composante normale du gradient du potentiel est nulle.

On recherche une solution de l'équation de Poisson sur le domaine carré $[0,1]\times[0,1]$. Sur les bords du carré, la condition limite appliquée consiste à fixer la valeur de u (condition limite de Dirichlet). La figure suivante montre un exemple de problème qui sera traité, avec une source nulle partout et avec deux lignes à l'intérieur du domaine où les valeurs de u sont imposées (modèle de condensateur plan). Un exemple de maillage utilisé pour la discrétisation est aussi représenté.

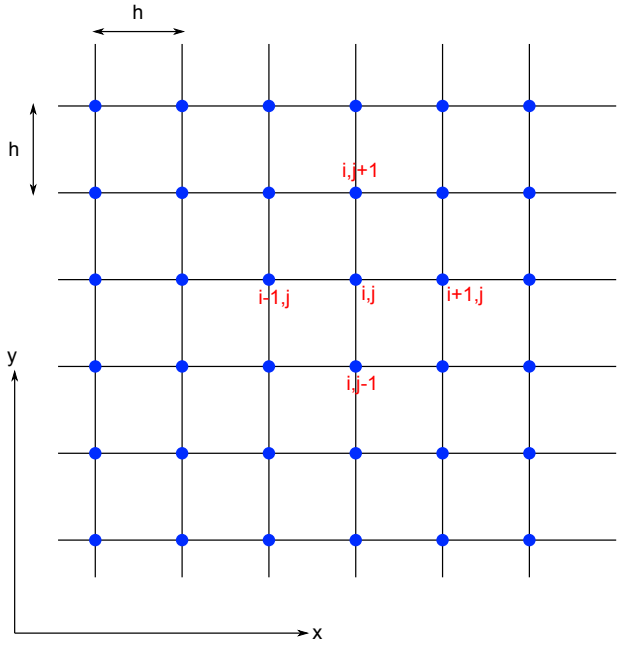


Nous allons voir comment effectuer la discrétisation de l'équation de Poisson, puis comment résoudre le système d'équations obtenu avec une méthode itérative.

2. Discrétisation

2.a. Maillage

Le domaine carré est divisé avec un maillage dont les mailles sont carrées. On note h le pas d'espace du maillage. La figure suivante représente quelques mailles.



On recherche une valeur approchée de la fonction $u(x, y)$ aux nœuds du maillage. La valeur recherchée au point d'indices (i, j) est notée $U_{i,j}$. La valeur de la fonction source au même point est notée $S_{i,j}$.

On a donc deux matrices U et S (on verra plus loin pour leurs dimensions).

2.b. Laplacien

Nous allons discrétiser l'équation de Poisson en utilisant la méthode des *différences finies*. Cette méthode consiste à remplacer les dérivées par des différences finies, afin d'obtenir des équations vérifiées par les valeurs $U_{i,j}$ aux nœuds du maillage.

Pour discrétiser une dérivée seconde, on utilise les développements de Taylor suivants, pour une fonction f d'une variable :

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3}f'''(x) + O(h^4) \tag{5}$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{3}f'''(x) + O(h^4) \tag{6}$$

En sommant ces deux équations, on obtient :

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + O(h^2) \tag{7}$$

Cette relation permet de discrétiser les dérivées secondes, par exemple :

$$\frac{\partial^2 U}{\partial x^2} \rightarrow \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h^2} \tag{8}$$

En procédant de même pour l'autre dérivée, on obtient l'équation suivante pour chaque nœud du maillage :

$$U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{i,j} = h^2 S_{i,j} \tag{9}$$

Cette équation constitue un schéma de discrétisation de l'équation de Poisson. Nous allons l'écrire sous une forme plus générale :

$$D_{i,j}U_{i+1,j} + G_{i,j}U_{i-1,j} + H_{i,j}U_{i,j+1} + B_{i,j}U_{i,j-1} + C_{i,j}U_{i,j} = F_{i,j} \tag{10}$$

Les matrices D (droite), G (gauche), H (haut), B (bas), C (centre) et S (source) ont les mêmes dimensions que le maillage. Par exemple, la matrice D comporte, pour chaque nœud du maillage, le coefficient par lequel il faut multiplier la valeur du nœud voisin situé à sa droite.

Pour un point non situé sur une frontière, on a donc :

$$\begin{aligned} D_{i,j} &= 1 \\ G_{i,j} &= 1 \\ H_{i,j} &= 1 \\ B_{i,j} &= 1 \\ C_{i,j} &= -4 \\ F_{i,j} &= h^2\, S_{i,j} \end{aligned}$$

2.c. Conditions limites de Dirichlet

Sur les bords du domaine, une valeur de u est imposée. Il peut aussi y avoir des lignes à l'intérieur du domaine où la valeur de u est imposée.

Pour un nœud (i, j) où la valeur imposée est U_0, l'équation s'écrit :

$$U_{i,j} = U_0 \tag{11}$$

L'équation générale (10) est donc valable à condition d'utiliser :

$$\begin{aligned} D_{i,j} &= 0 \\ G_{i,j} &= 0 \\ H_{i,j} &= 0 \\ B_{i,j} &= 0 \\ C_{i,j} &= 1 \\ F_{i,j} &= U_0 \end{aligned}$$

2.d. Dimensions du maillage

L'équation (10) a l'avantage de s'appliquer à tous les nœuds du maillage. Il existe d'ailleurs d'autres conditions limites qui peuvent se mettre sous cette forme. Elle peut même s'appliquer à d'autres équations linéaires à dérivées partielles.

On voit cependant un problème sur les bords du maillage. Par exemple sur le bord inférieur, il n'y a pas de point situé en dessous $(i, j-1)$. Afin d'éviter d'avoir à faire des tests pour savoir si le point est sur un bord, on ajoute des rangées fictives sur les bords. Ces rangées ne sont pas utilisées dans le calcul mais permettent d'adresser les nœuds en dehors du domaine.

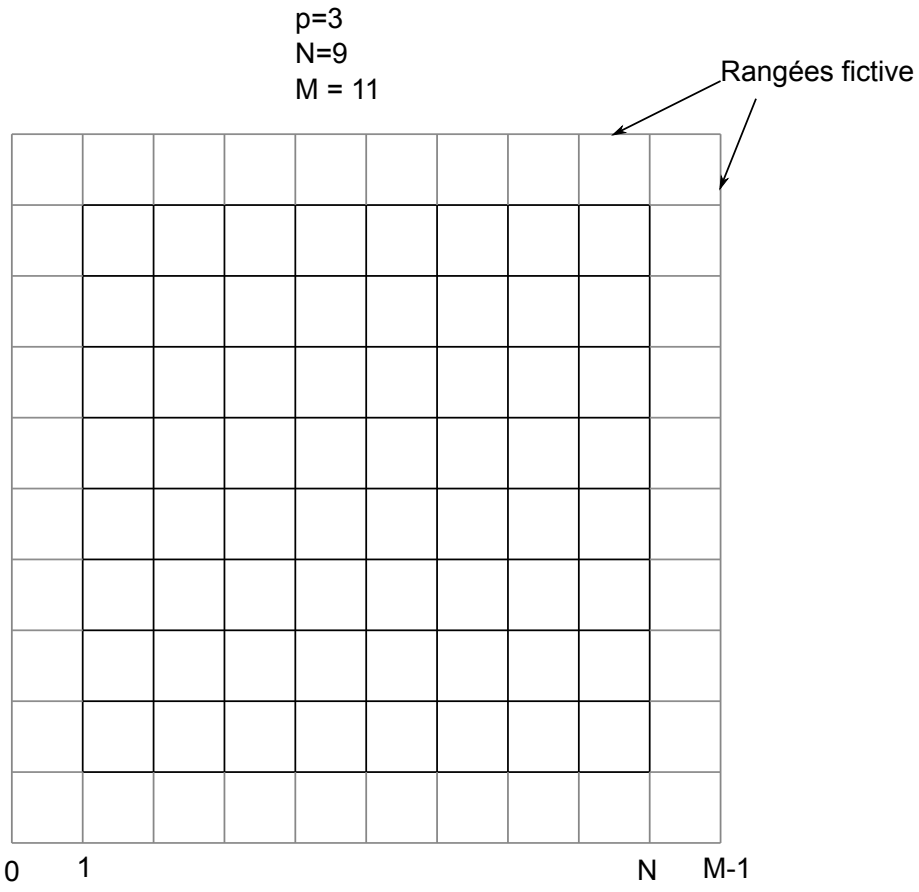
Par ailleurs, le nombre de nœuds dans chaque dimension sera choisi égal à une puissance de 2 plus 1 :

$$N = 2^p + 1 \tag{12}$$

où p est un entier. Cela fait un nombre de mailles puissance de 2. L'avantage de ce choix est de permettre une division du nombre de mailles par 2, 4, etc. En comptant les rangées fictives, le nombre d'éléments des matrices dans chaque dimension est :

$$M = N + 2 \tag{13}$$

La figure suivante montre un exemple avec $p = 3$, soit 8×8 mailles. Dans ce cas, les matrices U, G, D, H, B, F sont de dimension $(11, 11)$. Les indices d'accès aux éléments de la matrice sont indiqués sur la figure. On voit ainsi que les indices des nœuds du maillage vont de 1 à N.



3. Méthode itérative

3.a. Système d'équations linéaires

Pour chacun des $Q = N \times N$ nœuds du maillage, il y a une équation linéaire de la forme :

$$D_{i,j}U_{i+1,j} + G_{i,j}U_{i-1,j} + H_{i,j}U_{i,j+1} + B_{i,j}U_{i,j-1} + C_{i,j}U_{i,j} = F_{i,j} \tag{14}$$

On doit donc résoudre un système linéaire de Q équations à Q inconnues. Si on place toutes les inconnues $U_{i,j}$ dans une matrice colonne U (en les rangeant par exemple par lignes successives), le système peut se mettre sous la forme matricielle :

$$AU = F \tag{15}$$

La matrice A est carrée, de taille $Q \times Q$. Par exemple, pour $N = 65$, c'est une matrice de 4225×4225, soit environ 18 millions d'éléments. Il s'agit cependant d'une matrice creuse : chaque ligne de la matrice, correspondant à une équation, comporte seulement 5 éléments non nuls. Ce système peut en principe être résolu avec une méthode directe comme la méthode d'élimination de Gauss-Jordan.

Les méthodes directes sont cependant peu efficaces et peuvent présenter des problèmes de stabilité pour les très grands systèmes. On préfère utiliser une méthode itérative comme la méthode de Jacobi ou celle de Gauss-Seidel.

3.b. Méthode de Jacobi

La méthode de Jacobi est une méthode itérative ([1]) de résolution d'un système d'équations linéaires. Une méthode itérative consiste à partir d'une matrice $U^{(0)}$ comportant des valeurs initiales nulles, puis à appliquer une relation de récurrence pour calculer une suite de matrices $U^{(k)}$. La relation de récurrence doit être construite pour que la suite converge vers la solution du système d'équations :

$$\lim_{k\to\infty} U^{(k)} = U \tag{16}$$

Pour définir la méthode (et l'implémenter), il est plus simple de garder la numérotation des inconnues à deux indices, correspondant à la grille de discrétisation.

La matrice $U^{(k)}$ étant calculée, chaque élément de la matrice $U^{(k+1)}$ est calculé au moyen de la relation (14). La relation de récurrence de la méthode de Jacobi est donc :

$$U_{i,j}^{(k+1)} = \frac{1}{C_{i,j}}\left(F_{i,j} - D_{i,j}U_{i+1,j}^{(k)} - G_{i,j}U_{i-1,j}^{(k)} - H_{i,j}U_{i,j+1}^{(k)} - B_{i,j}U_{i,j-1}^{(k)}\right) \tag{17}$$

La méthode de Jacobi converge, c'est-à-dire que la suite ainsi construite tend vers une matrice limite. Cette limite est la solution du système linéaire, qui constitue la solution approchée de l'équation de Poisson aux points du maillage.

La méthode de Jacobi nécessite, à chaque itération, le calcul d'une nouvelle matrice. On lui préfère généralement la méthode de Gauss-Seidel, un peu plus efficace et ne nécessitant qu'une seule matrice.

3.c. Méthode de Gauss-Seidel

La méthode de Gauss-Seidel est une amélioration de la méthode de Jacobi, consistant à modifier la matrice en ligne, c'est-à-dire sans créer une nouvelle matrice.

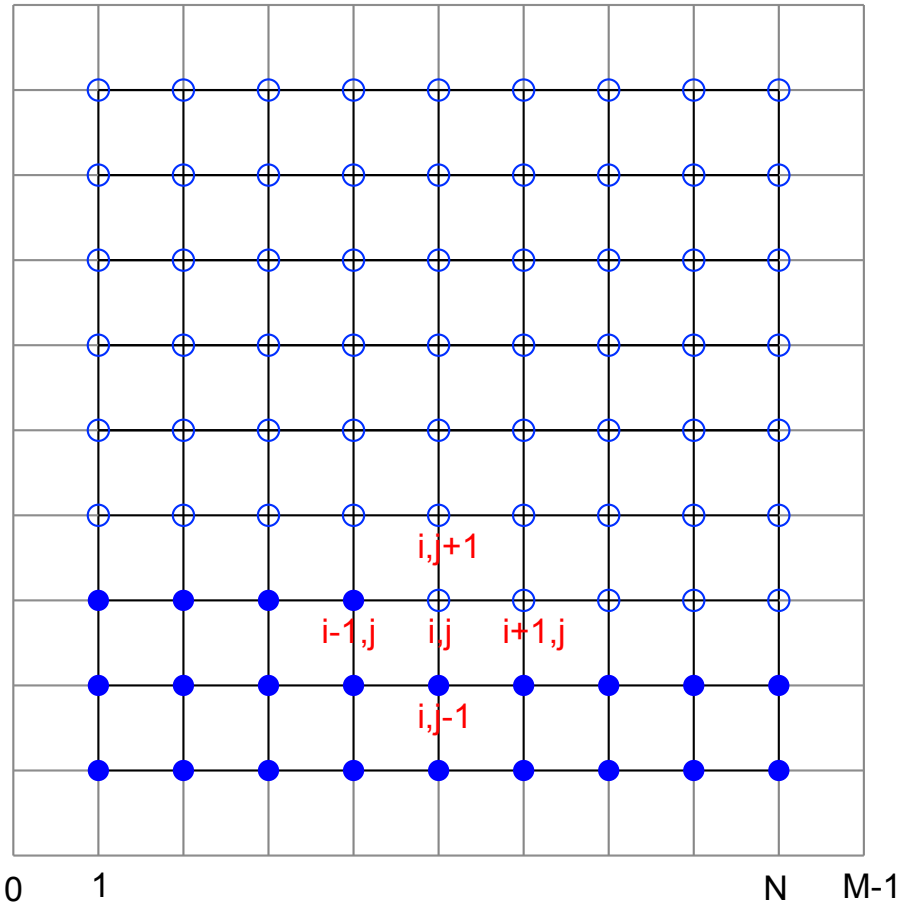
La matrice est parcourue par lignes successives, en partant de la ligne $j = 1$ (on utilise la numérotation précisée plus haut, avec les rangées fictives). Pour chaque ligne, l'indice de colonne i va de 1 à N. Le parcours complet de la matrice est appelé *itération de Gauss-Seidel*. Chaque élément de la matrice est modifié en utilisant ses 4 voisins, de la manière suivante :

$$U_{i,j} \leftarrow \frac{1}{C_{i,j}} \left(F_{i,j} - D_{i,j}U_{i+1,j} - G_{i,j}U_{i-1,j} - H_{i,j}U_{i,j+1} - B_{i,j}U_{i,j-1}\right) \qquad (18)$$

La modification de $U_{i,j}$ fait appel à deux points déjà calculés au cours de la même itération : le point situé à sa gauche et le point situé en dessous. La relation de récurrence s'écrit donc :

$$U_{i,j}^{(k+1)} = \frac{1}{C_{i,j}} \left(F_{i,j} - D_{i,j}U_{i+1,j}^{(k)} - G_{i,j}U_{i-1,j}^{(k+1)} - H_{i,j}U_{i,j+1}^{(k)} - B_{i,j}U_{i,j-1}^{(k+1)}\right) \qquad (19)$$

Pour l'implémentation, on utilise la relation (18). La figure suivante montre une itération en cours. Les cercles pleins indiquent les nœuds déjà traités par l'itération. Le nœud (i, j) est en cours de traitement.



Grace à l'ajout de rangées fictives sur les bords, le traitement des points des bords se fait aussi avec la relation (18). Par exemple, pour le bord du domaine situé à droite, il est possible de faire

appel à l'élément $U_{N+1,j}$ de la matrice, qui se trouve dans la rangée fictive. Cet élément est nulle mais il n'est pas utilisé car le nombre $D_{N,j}$ par lequel il est multiplié est nécessairement nul.

Pour suivre la convergence de la méthode de Gauss-Seidel, on calculera la norme suivante :

$$n(U) = \frac{1}{N^2}\sqrt{\sum_{i,j} U_{i,j}^2} \tag{20}$$

qui tend vers une limite finie.

3.d. Accélération par sur-relaxation

La méthode de Gauss-Seidel converge très lentement : le nombre d'itérations nécessaire pour obtenir une bonne approximation est proportionnel au nombre de points du maillage $Q = N^2$. Comme y a Q points à calculer pour chaque itération, cela donne un temps de convergence proportionnel à Q^2. C'est déjà mieux que la méthode d'élimination de Gauss-Jordan, qui est de complexité $O(Q^3)$.

La méthode de sur-relaxation permet d'accélérer considérablement la convergence. D'une manière générale, une méthode de relaxation consiste à modifier la méthode de Gauss-Seidel de la manière suivante :

$$U_{i,j} \leftarrow (1-\omega)U_{i,j} + \omega\left(\frac{1}{C_{i,j}}\left(F_{i,j} - D_{i,j}U_{i+1,j} - G_{i,j}U_{i-1,j} - H_{i,j}U_{i,j+1} - B_{i,j}U_{i,j-1}\right)\right) \tag{21}$$

où ω est le paramètre de relaxation. Si $0 < \omega < 1$, la nouvelle valeur est une moyenne pondérée de l'ancienne et de celle donnée par le schéma de Gauss-Seidel. Dans ce cas, la convergence est ralentie (sous-relaxation). La sur-relaxation consiste à choisir une valeur de ω strictement supérieure à 1. Pour que la méthode converge, il faut que :

$$1 < \omega < 2 \tag{22}$$

Il existe une valeur optimale du paramètre de relaxation, qui permet d'obtenir une convergence beaucoup plus rapide que la méthode de Gauss-Seidel, avec un nombre d'itérations proportionnel à N (la racine carrée du nombre d'inconnues). Cette valeur optimale dépend de l'équation discrétisée, des conditions limites, et de la taille du maillage. Pour l'équation de Poisson avec des conditions limites de Dirichlet sur les bords, l'expression du paramètre optimal est connue :

$$\omega^* = \frac{2}{1 + \sin\left(\frac{\pi}{N}\right)} \tag{23}$$

Lorsque N augmente, la valeur s'approche de 2. Par exemple, pour $N = 65$, la valeur optimale est $\omega^* = 1,91$. Pour cette valeur de N le gain par rapport à la méthode de Gauss-Seidel est déjà d'un facteur 65, ce qui signifie qu'un calcul complet durera une minute au lieu d'une heure. La complexité globale de la sur-relaxation optimale est $O(Q^{3/2})$, bien meilleure que la méthode directe de Gauss-Jordan de complexité $O(Q^3)$.

4. Travaux pratiques

4.a. Implémentation de la méthode de Gauss-Seidel avec sur-relaxation

```
import numpy
import math
from  matplotlib.pyplot import *
```

Les matrices U, G, D, H, B, F sont des tableaux `numpy.ndarray` de taille (M, M).
Pour faciliter l'implémentation, on définit une classe, dont on donne ici le constructeur :

```
class Poisson:
    def __init__(self,p):
        self.N = 2**p+1
        self.M = M = self.N+2
        self.h = 1.0/(self.N-1)
        self.C = numpy.zeros((M,M),dtype=numpy.float64)
        self.G = numpy.zeros((M,M),dtype=numpy.float64)
        self.D = numpy.zeros((M,M),dtype=numpy.float64)
        self.H = numpy.zeros((M,M),dtype=numpy.float64)
        self.B = numpy.zeros((M,M),dtype=numpy.float64)
        self.F = numpy.zeros((M,M),dtype=numpy.float64)
        self.U = numpy.zeros((M,M),dtype=numpy.float64)
        self.R = numpy.zeros((M,M),dtype=numpy.float64)
        self.omega = 2.0/(1.0+math.sin(math.pi*self.h))
```

Le constructeur calcule les nombres de points sur chaque dimension en fonction de l'entier p. Il crée les matrices et calcule aussi le pas d'espace h et le paramètre de relaxation optimal.
Pour les fonctions de la classe, on donne seulement l'entête :

```
    def laplacien(self):
```

effectue la discrétisation du laplacien, c'est-à-dire remplit les matrices C, G, D, H, B, F avec une source nulle ($F_{i,j} = 0$).

```
    def valeur_bords(self,v):
```

fixe la condition limite sur les bords, ce qui consiste à attribuer les bonnes valeurs de C, G, D, H, B, F sur les bords du domaine.

```
    def valeur_ligne_horiz(self,x,y,longueur,v):
```

fixe une condition limite de Dirichlet (valeur v imposée) sur une ligne horizontale située dans le domaine. Les coordonnées (x, y) du point de départ et la longueur sont données dans l'intervalle $[0, 1]$.

```
    def source_ligne_horiz(self,x,y,longueur,v):
```

définit une source sur une ligne horizontale, ce qui consiste à modifier les valeurs de F sur cette ligne.

```
    def source_ponctuelle(self,x,y,v):
```

définit une source ponctuelle, qui occupe un seul nœud du maillage.

```
    def norme(self):
```

calcule et renvoie la norme de la matrice U (équation (20)).

```
    def gauss_seidel(self,w1,w):
```

effectue une itération de Gauss-Seidel avec sur-relaxation, c'est-à-dire un parcours complet de la matrice U pour modifier ses valeurs selon le schéma (21). `w` est la valeur du paramètre de sur-relaxation ω et `w1` est $1-\omega$.

```
    def iterations(self,ni,w):
```

effectue `ni` itérations de Gauss-Seidel avec sur-relaxation. Après chaque itération, la norme de U est affichée avec `print`.

```
class Poisson:
    def __init__(self,p):
        self.N = 2**p+1
        self.M = M = self.N+2
        self.h = 1.0/(self.N-1)
        self.C = numpy.zeros((M,M),dtype=numpy.float64)
        self.G = numpy.zeros((M,M),dtype=numpy.float64)
        self.D = numpy.zeros((M,M),dtype=numpy.float64)
        self.H = numpy.zeros((M,M),dtype=numpy.float64)
        self.B = numpy.zeros((M,M),dtype=numpy.float64)
        self.F = numpy.zeros((M,M),dtype=numpy.float64)
        self.U = numpy.zeros((M,M),dtype=numpy.float64)
        self.R = numpy.zeros((M,M),dtype=numpy.float64)
        self.omega = 2.0/(1.0+math.sin(math.pi*self.h))

    def valeur_bords(self,v):
```

```
        for i in range(1,self.N+1):
            j = 1
            self.C[j][i] = 1.0
            self.G[j][i] = 0.0
            self.D[j][i] = 0.0
            self.B[j][i] = 0.0
            self.H[j][i] = 0.0
            self.F[j][i] = v
            self.U[j][i] = v
            j = self.N
            self.C[j][i] = 1.0
            self.G[j][i] = 0.0
            self.D[j][i] = 0.0
            self.B[j][i] = 0.0
            self.H[j][i] = 0.0
            self.F[j][i] = v
            self.U[j][i] = v
        for j in range(1,self.N+1):
            i = 1
            self.C[j][i] = 1.0
            self.G[j][i] = 0.0
            self.D[j][i] = 0.0
            self.B[j][i] = 0.0
            self.H[j][i] = 0.0
            self.F[j][i] = v
            self.U[j][i] = v
            i = self.N
            self.C[j][i] = 1.0
            self.G[j][i] = 0.0
            self.D[j][i] = 0.0
            self.B[j][i] = 0.0
            self.H[j][i] = 0.0
            self.F[j][i] = v
            self.U[j][i] = v

    def laplacien(self):
        for i in range(1,self.N+1):
            for j in range(1,self.N+1):
                self.C[j][i] = -4.0
                self.G[j][i] = 1.0
                self.D[j][i] = 1.0
                self.H[j][i] = 1.0
                self.B[j][i] = 1.0
                self.F[j][i] = 0.0

    def valeur_ligne_horiz(self,x,y,longueur,v):
        i0 = 1+int(x*(self.N-1))
        j0 = 1+int(y*(self.N-1))
        L = int(longueur*self.N)
        j = j0
        for i in range(i0,i0+L):
            self.C[j][i] = 1.0
            self.G[j][i] = 0.0
            self.D[j][i] = 0.0
            self.B[j][i] = 0.0
            self.H[j][i] = 0.0
            self.F[j][i] = v
            self.U[j][i] = v
```

```
    def source_ligne_horiz(self,x,y,longueur,v):
        i0 = 1+int(x*(self.N-1))
        j0 = 1+int(y*(self.N-1))
        L = int(longueur*self.N)
        h2 = self.h*self.h
        for i in range(i0,i0+L):
            j = j0
            self.F[j][i] = v*h2

    def source_ponctuelle(self,x,y,v):
        i0 = 1+int(x*(self.N-1))
        j0 = 1+int(y*(self.N-1))
        h2 = self.h*self.h
        self.F[j0][i0]=v*h2

    def zero(self):
        for j in range(2,self.N):
            for i in range(2,self.N):
                self.U[j][i] = 0.0

    def echelon(self,v):
        i1 = (self.N-1)/2
        for j in range(2,self.N):
            for i in range(2,i1):
                self.U[j][i] = v
            for i in range(i1,self.N):
                self.U[j][i] = 0.0

    def norme(self):
        z = 0.0
        for i in range(1,self.N+1):
            for j in range(1,self.N+1):
                z += self.U[j][i]*self.U[j][i]
        return math.sqrt(z)/(self.N)

    def gauss_seidel(self,w1,w):
        for j in range(1,self.N+1):
            for i in range(1,self.N+1):
                self.U[j][i] = w1*self.U[j][i]+w*(self.F[j][i]-self.G[j][i]*self.U[j][i-

    def iterations(self,ni,w):
        w1 = 1.0-w
        for k in range(ni):
            self.gauss_seidel(w1,w)
            print(self.norme())
```

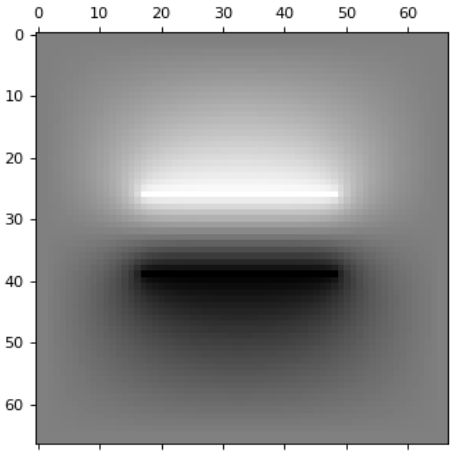
4.b. Exemples

Condensateur plan

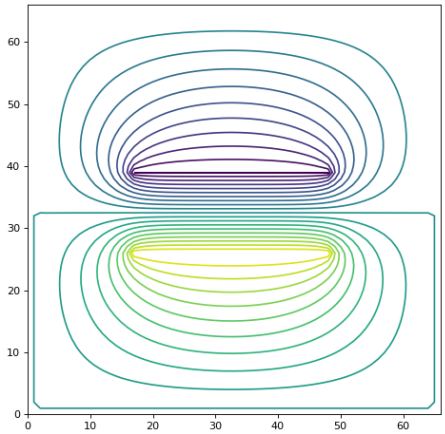
La géométrie a été définie en introduction. Il s'agit de deux plaques infinies dans la direction Z.

Voici un exemple avec $N = 65$. Une centaine d'itérations suffit.

```
poisson = Poisson(6)
poisson.laplacien()
poisson.valeur_bords(0.0)
poisson.valeur_ligne_horiz(0.25,0.4,0.5,1.0)
poisson.valeur_ligne_horiz(0.25,0.6,0.5,-1.0)
poisson.iterations(100,poisson.omega)
matshow(poisson.U,cmap='gray',vmin=-1.0,vmax=1.0)
```




```
figure(figsize=(7,7))
contour(poisson.U,20)
```



Fils chargés

Un fil chargé (parallèle à l'axe Z) est représenté par une source ponctuelle. Une source ponctuelle positive correspond à une charge négative. On pourra par exemple traiter le cas de deux fils de charges opposées :

```
poisson = Poisson(6)
```

```
poisson.laplacien()
poisson.valeur_bords(0.0)
poisson.source_ponctuelle(0.3,0.5,1)
poisson.source_ponctuelle(0.7,0.5,-1)
```

Références

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical recipes, the art of scientific computing*, (Cambridge University Press, 2007)