Exercices

Résolution numérique de l'équation de Laplace

Les équations de Maxwell permettent de montrer que, en régime stationnaire et dans le vide, le potentiel électrostatique V vérifie l'équation de Laplace, c'est-à-dire

$$\Delta V = 0$$

où Δ est l'opérateur laplacien scalaire.

Cette équation se retrouve dans de nombreux domaines : la température de tout système vérifie $\Delta T = 0$ en régime permanent et sans source de production, mais on la rencontre également en gravitation avec le potentiel associé au champ gravitationnel, en mécanique des fluides, pour le potentiel des vitesses, en mécanique quantique, etc.

Des solutions analytiques existent dans des cas simples, mais dans le cas général, résoudre cette équation demande d'avoir recours à des méthodes numériques : c'est le but du travail suit.

Nous nous limiterons à deux dimensions, ce qui revient physiquement à supposer une invariance par translation dans la troisième dimension.

Vous disposez du script Python `Eq_laplace_script-depart.py`, qui contient quelques fonctions qui nous serviront à l'initialisation des calculs et à l'affichage des résultats.

Ouvrir ce fichier et l'*enregistrer sous* un nom différent et dans *vos* documents.

1 - Principe de la résolution par méthode de relaxation

1 -.1 Schéma général

Il est évidemment impossible de résoudre numériquement l'équation de Laplace en tout point de l'espace : on se restreint à un domaine $\mathcal{D}$ fini, discrétisé sous forme d'une grille de dimensions $N_x \times N_y$.

On peut démontrer mathématiquement que l'équation de Laplace admet une unique solution dans un domaine fermé $\mathcal{D}$ si la valeur de V est fixée sur les bords B du domaine (on parle en mathématiques de « problème de Dirichlet »).

Dans notre cas, les bords correspondent au contour de la grille, mais peuvent aussi inclure d'autres points à l'intérieur de celle-ci sur lesquels le potentiel serait imposé, par exemple les armatures d'un condensateur.

Les conditions aux limites étant fixées, on utilise ensuite une méthode de résolution par différences finies.

L'idée est exactement la même que pour résoudre une équation différentielle par la méthode d'Euler : les dérivées spatiales du potentiel sont approximées par des différences entre les valeurs du potentiel entre deux points voisins de la grille.

La nouveauté par rapport à la méthode d'Euler est la résolution itérative : partant d'une « condition initiale » arbitraire (valeur de V fixée arbitrairement et « aléatoirement » en tout point de la grille), on améliore par récurrence la précision du résultat pour qu'il soit conforme à l'équation de Laplace.

1 -.2 Discrétisation de l'équation de Laplace

On rappelle qu'on se place sur une grille à deux dimensions de taille $N_x \times N_y$, schématisée figure 1. Le pas spatial de la grille est noté δ, si bien que le point (i, j) de la grille a pour coordonnées $(x_i, y_j) = (i\delta, j\delta)$. Les indices sont donc écrits dans tout ce travail dans une convention de

type « abscisse, ordonnée » plutôt que « ligne, colonne », comme ils le seraient pour un élément de matrice.

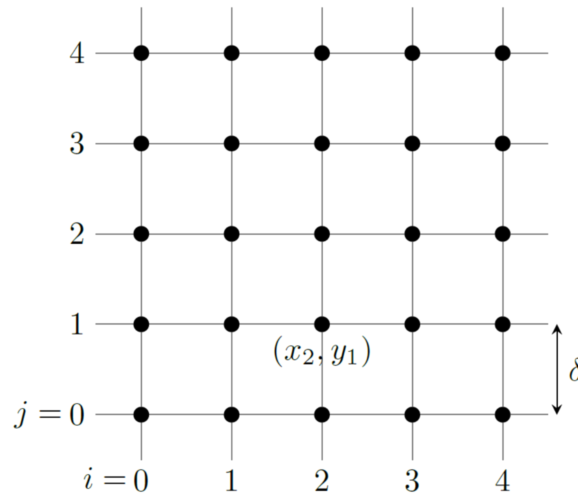


FIGURE 1 – Grille de discrétisation

À deux dimensions cartésiennes, le laplacien s'écrit

$$\Delta V = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2}$$

Il faut donc discrétiser la dérivée seconde sous forme de différence finie, ce qui se fait comme toujours grâce à des développements limités.

En transposant la formule de Taylor,

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2}f''(a) + O(h^3)$$

on peut écrire :

$$V(x_i + \delta, y_j) = V(x_i, y_j) + \delta\frac{\partial V}{\partial x}(x_i, y_j) + \frac{\delta^2}{2}\frac{\partial^2 V}{\partial x^2}(x_i, y_j) + O(\delta^3)$$

et de même

$$V(x_i - \delta, y_j) = V(x_i, y_j) - \delta\frac{\partial V}{\partial x}(x_i, y_j) + \frac{\delta^2}{2}\frac{\partial^2 V}{\partial x^2}(x_i, y_j) + O(\delta^3)$$

Le potentiel V sera stocké dans une variable globale sous forme d'un tableau `numpy` `V` tel que `V[i,j]` donne la valeur du potentiel $V(x_i, y_j)$.

Par simplicité de notation, on notera donc dans la suite $V[i,j] = V(x_i, y_j)$

1 - 1 - Montrer que

$$\frac{\partial^2 V}{\partial x^2}[i,j] \simeq \frac{V[i+1,j] + V[i-1,j] - 2V[i,j]}{\delta^2}$$

puis que

$$\Delta V[i,j] \simeq \frac{V[i+1,j] + V[i-1,j] + V[i,j+1] + V[i,j-1] - 4V[i,j]}{\delta^2}.$$

Correction

On fait les sommes :

$$V(x_i+\delta,y_j)+V(x_i-\delta,y_j)=2V(x_i,y_j)+\delta^2\left(\frac{\partial^2 V}{\partial x^2}\right)(x_i,y_j)+O(\delta^3)$$

D'où le résultat avec les notations proposées.

$$\frac{\partial^2 V}{\partial x^2}[i,j]\simeq\frac{V[i+1,j]+V[i-1,j]-2V[i,j]}{\delta^2}$$

Correction

De même pour la dérivée seconde par rapport à y.
On fait les sommes :

$$V(x_i,y_j+\delta)+V(x_i,y_j-\delta)=2V(x_i,y_j)+\delta^2\left(\frac{\partial^2 V}{\partial y^2}\right)(x_i,y_j)+O(\delta^3)$$

D'où le résultat avec les notations proposées.

$$\frac{\partial^2 V}{\partial y^2}[i,j]\simeq\frac{V[i,j+1]+V[i,j-1]-2V[i,j]}{\delta^2}$$

En sommant ces deux derniers résultats, soit les quatre termes calculés avec la formule de Taylor, on obtient, comme $\Delta V=\frac{\partial^2 V}{\partial x^2}+\frac{\partial^2 V}{\partial y^2}$:

$$\frac{\partial^2 V}{\partial x^2}[i,j]+\frac{\partial^2 V}{\partial y^2}[i,j]\simeq\frac{V[i+1,j]+V[i-1,j]+V[i,j+1]+V[i,j-1]-4V[i,j]}{\delta^2}$$

Ainsi, l'équation de Laplace discrétisée s'écrit, pour tout point (i,j) du domaine de résolution,

$$\Delta V[i,j]=0$$

Soit

$$\frac{V[i+1,j]+V[i-1,j]+V[i,j+1]+V[i,j-1]-4V[i,j]}{\delta^2}=0$$

ce que l'on peut encore mettre sous la forme

$$V[i,j]=\frac{V[i+1,j]+V[i-1,j]+V[i,j+1]+V[i,j-1]}{4}\tag{1}$$

Lorsqu'il vérifie l'équation de Laplace, le potentiel en un point est donc une moyenne des potentiels aux points voisins : c'est ainsi qu'il sera possible de construire par récurrence les valeurs de V en tout point de la grille en partant des bords où les conditions aux limites sont connues.

1 -.3 Définition des conditions aux limites

Rappelons que l'on appelle ici « bord » du domaine de résolution l'ensemble des points où la valeur du potentiel est imposée. Pour que la méthode utilisée s'applique, le bord doit nécessai-

rement inclure le contour géométrique de la grille mais peut également inclure d'autres points intérieurs.

Les points appartenant au bord du domaine sont stockés sous forme d'un tableau de booléens `B`, de taille $N_x \times N_y$.

Ce tableau est construit de la façon suivante :

— si le point (i, j) appartient au bord, alors `B[i,j]` vaut `True` : le potentiel en ce point est imposé, sa valeur ne doit jamais être modifiée.

— si le point (i, j) n'appartient pas au bord, alors `B[i,j]` vaut `False` : le potentiel en ce point est inconnu *a priori* mais doit vérifier l'équation (1).

Les tableaux décrivant le bord et le potentiel sont respectivement initialisés à des `False` et à des zéros.

2 - Écrire une fonction `initialisation_contour(V0)` qui prend comme argument un flottant `V0`. Cette fonction initialise les points du contour de la grille : elle leur affecte la valeur `True` dans `B` et la valeur `V0` dans `V` sans toucher aux autres points. Les variables `B` et `V` étant globales, la fonction les modifie mais ne retourne rien.

Correction

```
def initialisation_contour(V0):
    """ Initialise les bords de la grille a  la valeur V0 """
    V[:,0] = V0
    V[:,-1] = V0
    V[0,:] = V0
    V[-1,:] = V0
    B[:,0] = True
    B[:,-1] = True
    B[0,:] = True
    B[-1,:] = True
```

Exécuter cette fonction et, pour vérifier son bon fonctionnement, afficher les bords du domaine grâce à la fonction `graphe_bords()` que l'on trouve dans la rubrique des outils graphiques, au début du script fourni, et qui affiche en noir les points appartenant au bord du domaine et en blanc les autres (avec un dégradé grisé entre les deux). Choisir par exemple $N_x = N_y = 60$ pour obtenir un résultat visuel.

Correction

Dans la corps de programme principal

```
# Affichage du bord
initialisation_contour(0)
graphe_bords()
```

2 - Algorithme de Jacobi

2 -.1 Principe

L'algorithme de Jacobi permet une résolution itérative de l'équation (1) : la valeur du potentiel $V_n[i, j]$ au point $[i, j]$ à l'itération n se déduit de la valeur des potentiels voisins à l'itération $n - 1$

par

$$V_n[i,j] = \frac{V_{n-1}[i+1,j] + V_{n-1}[i-1,j] + V_{n-1}[i,j+1] + V_{n-1}[i,j-1]}{4} \quad (2)$$

Ainsi, la valeur du potentiel en tout point est recalculée à chaque itération en fonction de ce qu'elle aurait dû être si l'itération précédente vérifiait l'équation de Laplace. La convergence de la méthode est démontrable mathématiquement : au bout d'un nombre finies d'itérations, on est assuré d'approcher de la solution exacte et les nouvelles itérations ne font que très peu évoluer le potentiel.

Le nombre d'itérations dépend de la précision souhaitée. Dans ce travail, on utilise un critère basé sur l'écart quadratique moyen

$$e = \sqrt{\frac{1}{N_x N_y} \sum_{i,j} (V_{n+1}(i,j)) - V_n(i,j))^2} \quad (3)$$

qui représente l'écart entre la nouvelle et l'ancienne valeur du potentiel moyenné sur toute la grille. On choisit de stopper la simulation lorsque e devient inférieur à une valeur ε définie au début de la simulation, en fonction d'un compromis entre précision des résultats et temps de calcul.

2 -.2 Mise en pratique

L'algorithme de Jacobi est le suivant :

Initialisation :

→ créer le tableau `B` et le remplir avec des `True` et des `False` pour décrire les bords du domaine

→ créer le tableau `V` et l'initialiser avec les valeurs voulues sur les bords du domaine.

Itérations : actualiser le tableau `V` : pour tout point (i,j) n'appartenant pas au bord la nouvelle valeur est calculée selon la relation de récurrence (2).

Terminaison : calculer après chaque itération l'écart e à la précédente avec l'équation (3) et cesser la procédure lorsque e devient inférieur à ε.

3 - Écrire une fonction `ecart(V1,V2)` prenant en argument deux tableaux `numpy` et renvoyant l'écart entre ces deux tableaux au sens de l'équation (2).

Correction

```
def ecart(V1,V2):
    """ les tableaux ne sont pas forcément carrés
    """
    Nx = V1.shape[0]
    Ny = V1.shape[1]
    diff_carre = 0
    for i in range(Nx):
        for j in range(Ny):
            diff_carre += (V1[i,j]-V2[i,j])**2
    return (diff_carre/(Nx*Ny))**(1/2)
```

4 - Écrire une fonction `iteration_jacobi()` qui effectue une itération de l'algorithme ci-dessus. Cette fonction ne prend aucun argument et doit renvoyer l'écart e entre les potentiels avant et après itération.

Correction

```
def iteration_jacobi():
    '''renvoie le nouveau potentiel actualise si le point fait
  parti
        du domaine D a l exclusion du bord '''
    V_copie = V.copy()
    for i in range(Nx):
        for j in range(Ny):
            if B[i,j]: # si [i,j] est un point du bord ... ne rien
  faire
                V[i,j] = V_copie[i,j]
            else:
                V[i,j] = (V_copie[i+1,j]+V_copie[i-1,j]+V_copie[i,
  j+1]+V_copie[i,j-1])/4
    return ecart(V_copie,V)
```

Rappels utiles :

Pour effectuer une copie d'un tableau `numpy` on utilise `V_copie = V.copy()` mais pas `V_copie = V`, sans quoi les deux sont liés et toute modification de `V` entraîne une modification de `V_copie`.

Avoir choisi un tableau de booléens pour `B` permet d'écrire directement les tests sous la forme `if B[i,j]:`, qui est exactement équivalent à écrire `if B[i,j] == True:`, et même `if B[i,j] == 1:` car en Python les booléens sont équivalents à des 0 et des 1.

5 - Écrire une fonction `jacobi(eps)` qui prend en argument un flottant `eps` et qui itère l'algorithme tant que l'écart est supérieur à `eps`. À des fins de comparaison avec la méthode suivante, votre fonction doit renvoyer le nombre d'itérations. Pour suivre la convergence en temps réel, on pourra ajouter un `print(e)` à un endroit bien choisi.

Correction

```
def jacobi(eps):
    ''' actualise le tableau V donc inutile de le renvoyer
    '''
    compteur = 0
    while iteration_jacobi() > eps:
        iteration_jacobi()
        compteur += 1
        print(iteration_jacobi() - eps)
    return compteur
```

2 -.3 Application au condensateur plan de taille fini

6 - Représenter les surfaces équipotentielles d'un condensateur plan de taille finie, en procédant de la façon suivante :

1. Choisir une grille de taille 60×60 et créer les tableaux `B` et `V` dont toutes les valeurs sont nulles.
2. Initialiser le contour de la grille à un potentiel nul
3. Ajouter un condensateur dont les armatures sont aux potentiels ± 10 V avec la fonction `initialisation_condensateur(V1,V2)` qui prend en argument `(V1,V2)` respectivement les tensions de la plaque inférieure et supérieure.
4. Résoudre l'équation de Laplace par l'algorithme de Jacobi en choisissant comme critère de terminaison $\varepsilon = 1 \cdot 10^{-3}$.
5. Afficher les surfaces équipotentielles avec la fonction `graphe_equipot()`.

Correction

Dans le corps principal du programme. N'oubliez pas de mettre en commentaire (CTRL 3 Commenter/Décommenter) la partie du programme principal qui n'est pas utile.

```
initialisation_contour(0)
initialisation_condensateur(-10,10)
print(jacobi(1e-3))
graphe_equipot()
```

7 - Identifier les effets de bord et la distance sur laquelle ils se manifestent. Comme vous le savez les équipotentielles d'un condensateur plan infini sont équidistantes, parallèles entre elles, et parallèles aux armatures du condensateur.

Correction

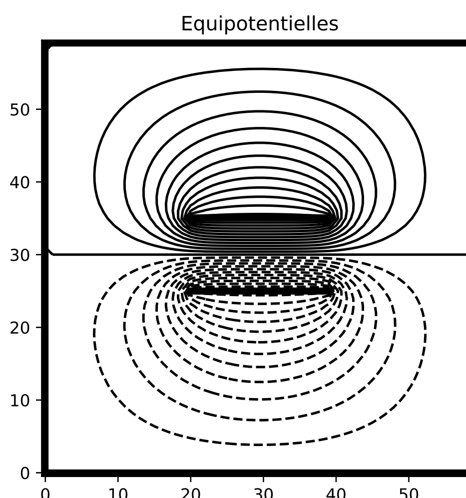


FIGURE 2 – Équipotentielles d'un condensateur

Les effets de bord se font sentir, près des bords où les équipotentielles s'écartent.

3 - Algorithme de Gauss-Seidel adaptatif

On peut montrer que la complexité de l'algorithme de Jacobi pour une grille de taille $N \times N$ est en $O(N^3)$, ce qui est néfaste pour les temps de calcul et devient rapidement gênant, même pour des géométries simples. L'algorithme de Gauss-Seidel adaptatif est une amélioration possible, mais moins immédiatement compréhensible.

3 -.1 Principe

La méthode consiste essentiellement à remplacer la relation de récurrence (2) par

$$V_n[i,j] = (1-\omega)V_{n-1}[i,j] + \omega \times \frac{V_{n-1}[i+1,j] + V_n[i-1,j]V_{n-1}[i,j+1] + V_n[i,j-1]}{4} \quad (4)$$

Il y a donc deux différences majeures par rapport à l'algorithme précédent.

D'une part, on utilise la valeur $V_{n-1}[i,j]$ de V au point (i,j) à l'itération précédente, pondérée avec un poids ω.

On peut montrer que pour une grille carrée de taille $N \times N$ il existe une valeur optimale

$$\omega_{ropt} = \frac{2}{1+\frac{\pi}{N}}$$

D'autre part, le calcul fait intervenir les valeurs du potentiel aux points voisins à l'itération précédente $n-1$ comme c'était déjà le cas, mais aussi certaines valeurs à l'itération n elle-même. Ceci est rendu possible par le fait que la grille est parcourue à i et j croissants, ces valeurs ont donc déjà été actualisées lors de l'itération n : on comprend ainsi que cela améliore la vitesse de convergence. En pratique, il ne faut plus utiliser un tableau copie mais directement itérer V avec l'équation (4).

8 - Écrire les fonctions `iteration_gauss_seidel()` et `gauss_seidel(eps)` qui jouent un rôle analogue aux fonctions du même nom écrites pour l'algorithme de Jacobi.

Correction

```
def iteration_gauss_seidel():
    '''renvoie le nouveau potentiel actualise si le point fait parti
        du domaine D a l exclusion du bord '''
    V_copie = V.copy()
    omega = 2/(1+np.pi/Nx)
    for i in range(Nx):
        for j in range(Ny):
            if B[i,j]: # si [i,j] est un point du bord ... ne rien faire
                V[i,j] = V_copie[i,j]
            else:
                V[i,j] = (1-omega)*V_copie[i,j]+omega*(V_copie[i+1,j]+V[i-1,j]+V_copie[i,j+1]+V[i,j-1])/4
    return ecart(V_copie,V)
```

Correction

```
def gauss_seidel(eps):
    ''' actualise le tableau V donc inutile de le renvoyer
    '''
    compteur = 0
    while iteration_gauss_seidel() > eps:
        iteration_gauss_seidel()
        compteur += 1
        print(iteration_gauss_seidel() - eps)
    return compteur
```

9 - Tester ces fonctions sur l'exemple précédent du condensateur. Vérifier que le nombre d'itérations est inférieur.

Pour que la comparaison ait un sens, penser à réinitialiser la matrice V entre les deux simulations.

Correction

Pour l'algorithme de Jacobi, on trouve 166 itérations contre 27 pour Gauss-Seidel.

```
initialisation_contour(0)
#graphe_bords()
initialisation_condensateur(-10,10)
print(gauss_seidel(1e-3))
graphe_equipot()
```

3 -.2 Taille du domaine de calcul

La taille du domaine de calcul est un point crucial dans les simulations numériques de ce type. Il est nécessaire, pour la simulation, de fixer la valeur du potentiel sur tout le contour de la grille. Cette nécessité de la modélisation ne doit pas ou peu influer sur le résultat de la simulation dans la zone intéressante.

On étudie cet effet dans le cas de domaines carrés de taille $N \times N$, les différentes valeurs étudiées étant stockées dans une liste `lst_N`. En guise d'illustration, on étudie l'influence de la valeur de N sur le potentiel en un point de référence choisi arbitrairement, légèrement à l'extérieur du condensateur. Les valeurs du potentiel en ce point pour les différentes valeurs de N sont stockées dans une liste `Vref`. On ne pourra prendre que des valeurs $N \geq 30$ compte tenu du point choisi.

10 - Compléter la fonction `influence_taille_domaine(lst_N)`, qui renvoie la liste `Vref`. On notera que quelques précautions s'imposent avec les variables choisies comme étant globales dans les simulations précédentes et dont cette fonction modifie les valeurs.

Correction

Comme nous devons redéfinir les grilles à chaque fois, les variables ont été précisées en entête de la fonction. La priorité sera donc donné aux valeurs définies dans la fonction.

```python
def influence_taille_domaine(lst_N):
    global B, V, Nx, Ny
    Vref = []
    for N in lst_N:
        print(N)
        Nx, Ny = N, N
        V = np.zeros((Nx,Ny))
        B = np.zeros((Nx,Ny), dtype=bool)
        initialisation_contour(0)
        initialisation_condensateur(-10,10)
        gauss_seidel(1e-3)
        Vref.append(V[Nx//2+2,Ny//2+14])
    return Vref
```

11 - Représenter le potentiel au point de référence en fonction de la taille du domaine de calcul.

Correction

```python
lst_N=[30,40,50,60,70,80,90,100,110,120,150]
Pot = influence_taille_domaine(lst_N)

plt.figure()
plt.plot(lst_N,Pot,marker = '+')
plt.xlabel('N : taille du domaine')
plt.ylabel('Potentiel au même point')
plt.show()
```

Correction

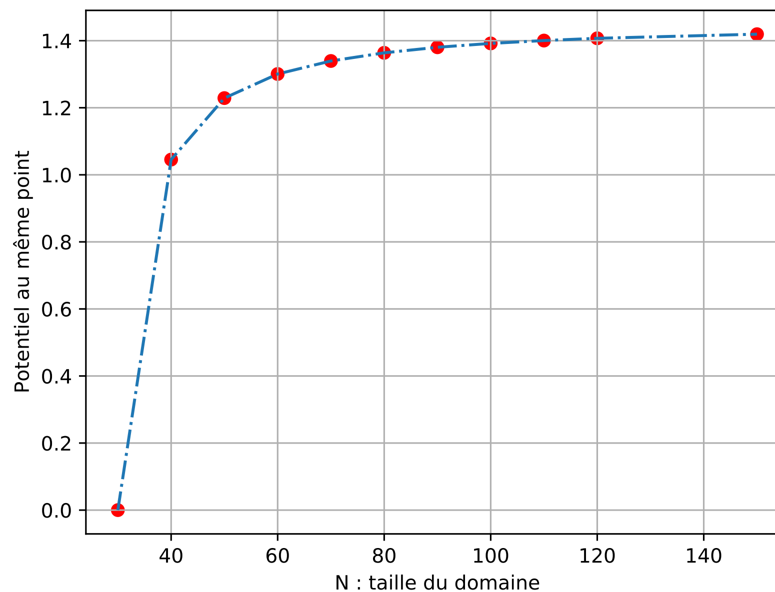


FIGURE 3 – Influence de la taille du domaine sur la convergence du potentiel en un point

3 -.3 Application à l'effet de pointe

Introduction

En l'absence de perturbations, le potentiel électrostatique augmente quasi-linéaire dans l'atmosphère entre le sol – sol qui est à un potentiel `Vsol` que l'on prendra égal à 0 V) – et l'ionosphère. Dans une atmosphère non perturbée, par temps calme, le gradient de potentiel est de $100\,\mathrm{V\cdot m^{-1}}$ environ.

L'ajout d'une tige verticale conductrice, un paratonnerre par exemple, vient perturber cette situation : ses propriétés conductrices font qu'elle est au même potentiel en tout point, égal à celui du sol auquel elle est reliée.

Ceci va déformer les surfaces équipotentielles, et donc modifier le champ électrique. On s'attend à ce que ce dernier augmente vers la pointe : c'est ce que l'on nomme l'effet de pointe, et qui explique que la foudre tombe préférentiellement sur les objets pointus.

On simule ici une tige verticale de hauteur H et de largeur D, dont la pointe est une demi-sphère de diamètre D, voir figure (4).

Plus précisément, notre programme étant bidimensionnel, nous simulons en fait un plan de hauteur H, épaisseur D, de longueur très grande dans la direction perpendiculaire au plan de la simulation.

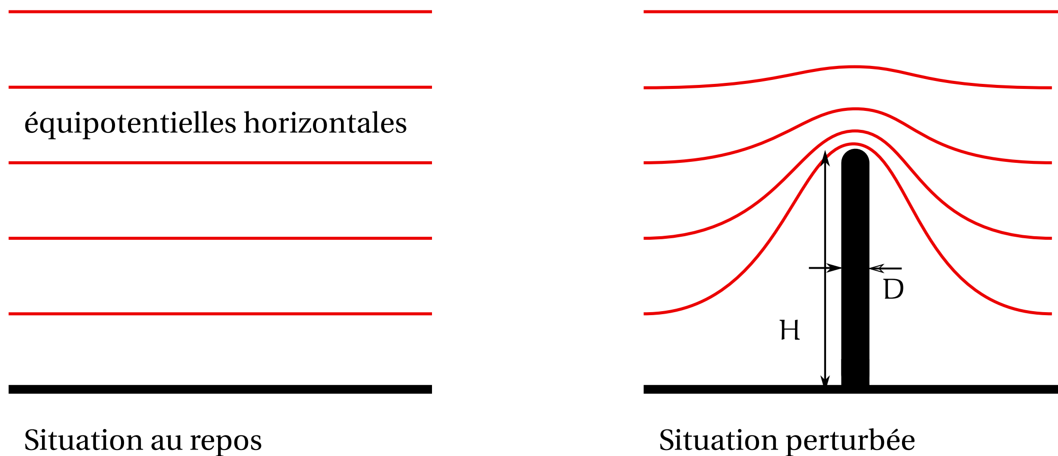


FIGURE 4 – Allure qualitative des équipotentielles en présence d'une pointe.

L'objectif est d'obtenir numériquement une relation entre la valeur E_{max} du champ électrique et le diamètre D de la pointe de la tige.

Calcul du potentiel

La fonction `initialisation_effet_pointe(Vmin,Vmax,h,d)` fournie dans le script de départ permet d'initialiser un domaine avec :

1. un contour dont le bord bas est au potentiel V_{min}, le bord haut au potentiel V_{max} et où le potentiel augmente linéairement de V_{min} à V_{max} sur les bords droit et gauche.
2. une tige de hauteur h et épaisseur d (en nombre de points), au potentiel fixé V_{min}. Attention, le nombre d doit être impair !

On raisonnera sur un pas de grille $\delta = 3$ cm, et on prendra :

1. $h = 50$ et $d = 13$, ce qui correspond à une tige de hauteur $H = 1,5$ m et de largeur $D = d \times \delta = 30$ cm
2. un domaine de taille 120×120 points, ce qui correspond à 3,6 cm $\times$ 3,6m
3. un potentiel $V_{min} = 0$ V au niveau du sol et $V_{max} = N_x \times \delta \times 100$ V en haut du cadre, ce qui donne bien un gradient au repos de 100 $\mathrm{Vm^{-1}}$.

12 - Initialiser le problème à l'aide de cette fonction. Vérifier que tout est correct à l'aide de `graphe_bords()`.

Correction

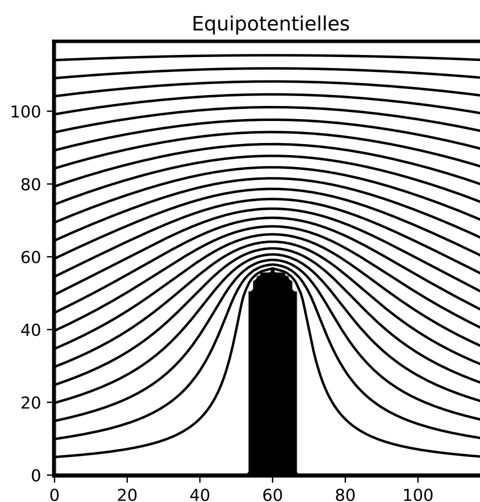
```
delta = 3
Nx = 120
Ny = 120
h = 50
d = 13

Vmin =0
Vmax = Nx * delta * 100

V = np.zeros((Nx,Ny))
B = np.zeros((Nx,Ny), dtype=bool)

initialisation_effet_pointe(Vmin,Vmax,h,d)
graphe_bords()
```

13 - Utiliser ensuite l'algorithme de Gauss-Seidel pour obtenir le potentiel. On prendra encore $\varepsilon = 10^{-3}$. Visualiser la solution à l'aide de `graphe_equipot()`.

Correction

```
initialisation_effet_pointe(Vmin,Vmax,h,d)
#graphe_bords()
gauss_seidel(1e-3)
graphe_equipot()
```

Correction



Calcul du champ électrique

14 - Sur les équipotentielles précédentes, identifier la zone où le champ est le plus intense.

Pour calculer le champ électrique, on évaluera les dérivées premières par des relations du type

$$\frac{\partial V}{\partial x}(x_i, y_j) = \frac{V(x_i+\delta, y_j) - V(x_i-\delta, y_j)}{2\delta}$$

qui donne une précision sur la dérivée en $O(\delta^2)$, d'un ordre meilleure que celle du schéma d'Euler habituel.

15 - - Écrire une fonction `calcul_champ(delta)` qui prend comme argument le pas de la grille et renvoie un tableau `norme_E` de taille $N_x \times N_y$ qui contient la norme du champ électrique. On ne cherchera pas à calculer le champ sur les bords du domaine.

Correction

```
def calcul_E(delta):
    Ex = np.zeros((Nx,Ny))
    Ey = np.zeros((Nx,Ny))
    norme_E = np.zeros((Nx,Ny))
    for i in range(1,Nx-1): # on évite les bords
        for j in range(1,Ny-1): # on évite les bords
            Ex[i,j] = -(V[i+1,j]-V[i-1,j])/(2.*delta)
            Ey[i,j] = -(V[i,j+1]-V[i,j-1])/(2.*delta)
            norme_E = (Ex**2+Ey**2)**0.5
    return Ex, Ey, norme_E
```

16 - Appliquer cette fonction dans le cas précédent : par combien la présence de l'obstacle multiplie-t-elle le champ ?

Correction

```
Ex, Ey, normeE = calcul_E(delta)
Emax = np.max(normeE)

# on trouve 530 .. donc fois 5.3
```

17 - Reproduire la procédure pour quelques valeurs de d, par exemple avec $d \in [7, 13, 17, 23]$ et représenter le maximum de la norme de $\vec{E}$, E_{max} en fonction de d. Ceci va-t-il dans le sens de l'effet de pointe mentionné précédemment ?

Correction

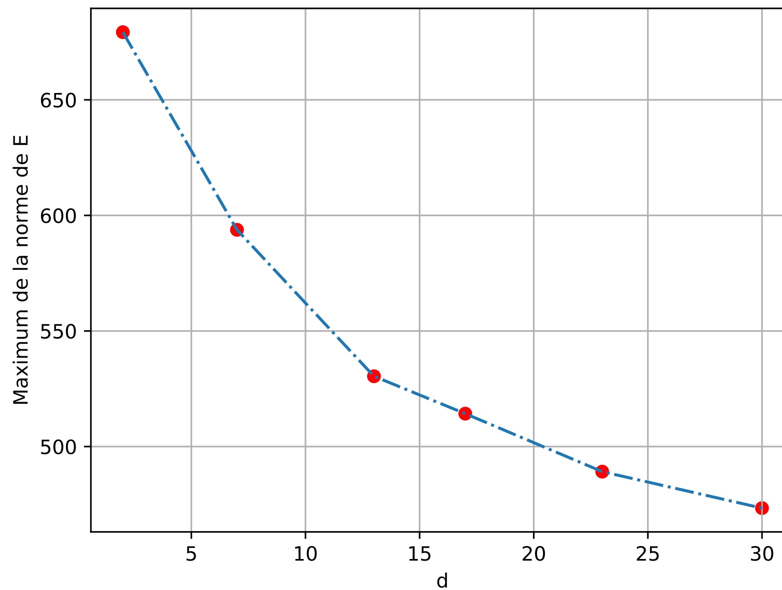


FIGURE 5 – E_{max} en fonction de d, diamètre de la pointe

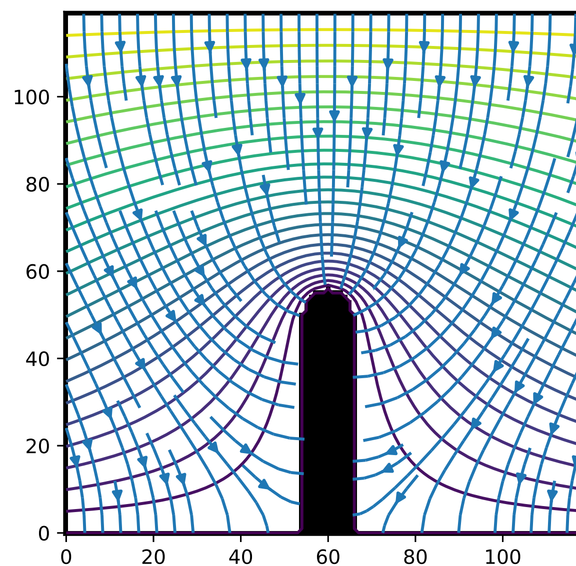


FIGURE 6 – Lignes de champ et équipotentielles autour d'une pointe