Poly de TP de physique numérique, UE LU3PY126 FOAD

26 janvier 2023

Sorbonne Université - L3 Physique

Table des matières

TP 0

Marches aléatoires

0.1 Nombres aléatoires

Pour simuler informatiquement un processus aléatoire, il est nécessaire d'utiliser un générateur qui fournit une suite de nombres s'approchant le plus possible d'un «vrai» hasard. Par vrai hasard on entend par exemple le fait que la suite produite ne doit pas être périodique et que les nombres produits doivent être uniformément répartis.

On utilise en pratique des algorithmes qui produisent une suite périodique, mais avec une période très grande : par exemple $m = 2^{32}$. Un exemple d'algorithme utilisé pour générer une suite X_n de nombres est :

$$X_{n+1} = (aX_n + c) \bmod m$$

où le modulo m détermine la période maximale de la suite et X_0 est la "graine" (ou "seed" en anglais), le premier terme de la suite. Ce générateur de nombre aléatoires (random number generator, RNG) s'appelle "générateur congruentiel linéaire" (linear congruential generator (LCG)) inventé par le mathématicien américain Derrick Lehmer en 1948 pour c=0 (Lehmer RNG). On peut tester cet algorithe en calculant à la main la suite des X_n pour une valeur faible de m :

itération	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	$\cdots$
$m = 9, a = 4$,c=1	0	1	2	5	3	4	8	6	7	2	0	1	$\cdots$
$m = 9, a = 2$,c=0	1	2	4	8	7	5	1	2	4	8	7	5	$\cdots$

1. Vérifiez rapidement ces calculs. Quelle est la périodicité des deux suites obtenues ?

0.1.1 Générer des nombres aléatoires avec NumPy

En python, numpy contient déjà tout ce qu'il faut pour générer des nombres aléatoires ("random numbers")

```
import numpy as np
# On initialise le "random number generator" avec la graine 12345 :
np.random.seed(12345)
for i in range(10):
    print(np.random.rand()) # un réel dans [0,1)
for i in range(10):
    print(np.random.random_integers(low=1, high=6)) # un entier dans
    ↪ [1,6]
```

2. Exécutez le code ci-dessus plusieurs fois. Puis, Enlevez l'appel à `np.random.seed()` et exécutez le code modifié plusieurs fois. Qu'est-ce que vous observez ?

Les fonctions `np.random.rand()` et `np.random.random_integers()` renvoient la même suite de nombres à chaque exécution si on ne change pas la graine X_0. Essayez de comprendre pourquoi il est souhaitable et important, en programmation scientifique, que les séquences de nombres aléatoires ne changent pas d'une exécution à l'autre du programme.

En utilisant l'argument `size=N` ces fonction renvoient N nombres aléatoires sans passer par une boucle `for`. À l'aide de la fonction `np.random.rand()`, il est facile d'obtenir des nombres flottants dans l'intervalle que l'on souhaite qui pourront être stockés dans un `tableau numpy`. :

```
N = 10
a = np.random.rand(size=N)     # a contient N  flottants aléatoires avec
↪  0 <= a < 1
b = 3.0 * np.random.rand(size=N) + 6.5; # b contient N  flottants
↪  aléatoires avec 6.5 <= b < 9.5
```

3. Ecrivez un code qui simule le lancé de deux dés à six faces (non truqués !) jusqu'à ce que l'on obtienne un "double six". Utilisez une boucle `while`.

0.2 Marches aléatoires

Les marches aléatoires sont le point de départ de nombreuses modélisations physiques décrivant le mouvement brownien, la diffusion, les polymères, etc. Elles ont été introduites en physique en 1905 par Albert Einstein.

0.2.1 Marches aléatoires à pas discrets

Le modèle le plus simple d'une marche aléatoire à une dimension est celui d'une particule pouvant se déplacer par pas discrets (± 1) qui s'effectuent au hasard à droite ou à gauche (voir Figure 0.1). La particule part de l'origine ($x = 0$) et la marche s'arrête au bout d'un nombre n de pas. La marche est dite isotrope si la probabilité d'aller à droite est la même $p = q = 1/2$ que celle d'aller à gauche.

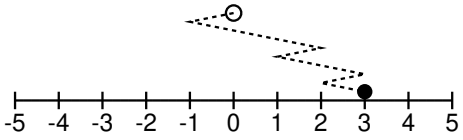


FIGURE 0.1 – Une marche aléatoire discrète à 1D.

4. Comment feriez-vous pour réaliser un tirage "à pile ou face" ? Le résultat du tirage sera représenté par les nombres entiers relatifs ± 1 ?

Il y a plusieurs chemins qui mènent à Rome, surtout en python. Voici quelques solutions possibles :

- `np.random.choice([-1,1], size=N)` est une solution simple que vous pouvez utiliser, voir l'aide dans Spyder sur cette fonction (clique sur la fonction puis Ctrl+I).

```
N= 10
a = np.random.choice([-1,1], size=N)
```

- On génère un ensemble de nombres aléatoires entiers entre zéro et un inclut, donc au final des zéros et des uns, et on les stocke dans un tableau nommé `x` par exemple. Ensuite on applique la formule `2*x - 1` d'une façon vectorielle sur tous les éléments de x, afin d'avoir que des -1 et des 1. Donc pas besoin de faire des tests pour filtrer des zéros, grâce à cette formule.

```
x = np.random_integers(0,1, size=N)
a = 2*x -1
```

- `np.random.rand()` peut être utilisé avec un test `if` dans une boucle :

```
a = np.empty(N) # on crée un tableau numpy
for i in range(N):
    if np.random.rand() < 0.5 :
        a[i] = 1
    else:
        a[i] = -1
```

- Il est également possible d'utiliser `x=np.random.rand(N)` sans boucle, avec la formule `2*(x<0.5)-1` qui donne -1 ou +1 aléatoirement, car `x<0.5` sera =0 si la condition est fausse ou =1 si la condition est vraie :

```
x = np.random.rand(size=N)
a = 2 * (x<0.5) -1
```

5. Écrivez un programme qui fait la simulation d'**une** marche de N pas. La position du "marcheur" est l'intégrale, ou la somme cumulée des N pas. Créez une boucle sur sur N pas et stockez la position en fonction du temps. Affichez la position en fonction du temps, ainsi que la position finaleen utilisant la fonction `plot()` de `matplotlib`.

```
import numpy as np
import matplotlib.pyplot as plt

#np.random.seed(12345)
NPAS = 100
marche = np.empty(NPAS) # un array numpy vide pour stocker la position
↪  en fonction du temps

def pile_face():
    return ... # Choisissez une approche parmi celles proposée ci-dessus

marche[0] = 0 # position initiale
for i in range(1,NPAS):
    pas = pile_face()
    marche[i] = ... # Calculez la somme cumulée

# Affichage des résultats
plt.plot(np.arange(NPAS),marche)
```

```
# TRES IMPORTANT ! Bien définir les axes de tout graphe
plt.title('position finale : %d' % marche[-1])
plt.xlabel('# pas')
plt.ylabel('position')
```

Les boucles sont lentes en `python` et une approche pour accélérer le programme est de le vectoriser. L'idée consiste à remplacer une (ou plusieurs) boucle(s) par des calculs sur un `tableau numpy`. Par exemple ici, au lieu de choisir une par une les valeurs aléatoires du pas, on va créer directement un `tableau` contenant tous les pas choisis aléatoirement et faire les calculs directement en utilisant ce tableau. (Voir Annexe 0.3.1)

0.2.2 Analyse statistique

Afin de caractériser les propriétés statistiques de ce modèle, il est nécessaire d'effectuer un grand nombre de marches, toutes du même nombre N de pas.

6. Modifiez ce programme en définissant une **fonction** `marche(npas)` qui effectue une marche aléatoire de `npas` pas et renvoie la position finale. Utilisez maintenant cette fonction pour écrire un programme pour effectuer un nombre `NMARCHES` de marches de `npas` pas chacune. Chaque marche est indépendante de la précédente, et redémarre donc depuis l'origine. Calculez la moyenne sur toutes ces `NMARCHES` marches de la position d'arrivée du marcheur, ainsi que son écart type. en utilisant les fonctions `mean()` et `std()` de `numpy`.

```
import numpy as np

#np.random.seed(12345)
NPAS = 100
NMARCHES = 1000

arrivee = np.empty(NMARCHES) # array numpy vide pour stocker les
↪  NMARCHES positions d'arrivée

def pile_face(npas):
    return ...

def marche(pas):
...
for i in range(NMARCHES):
    arrivee[i] = marche(NPAS)

print('<x> = %.2g, ectat-type = %.4g' %
↪  (np.mean(arrivee),np.std(arrivee)))
```

Dans le cadre de cette étude, le nombre de positions x demeure peu élevé et donc le stockage en mémoire de toutes les positions dans un `tableau numpy` avant d'effectuer le calcul de la moyenne et de l'écart-type ne pose pas de problème. Si *a contrario* la quantité de données devient très importante et que seules nous intéressent la moyenne et l'écart-type (ou la valeur quadratique moyenne), il est alors intéressant d'utiliser leur définition pour les calculer "à la volée", c'est-à-dire en les mettant à jour à chaque itération de la boucle (voir l'annexe 0.3.2).

7. Faîtes les mêmes simulations pour un nombre de pas variant de $n = 50$ jusqu'à $n = 6400$ en doublant le nombre de pas à chaque fois : c'est-à-dire effectuez `NMARCHES` de $n = 50$ pas, en calculant les mêmes quantités statistiques qu'au point précédent, puis `NMARCHES` de $n = 100$ pas, etc.... Tracer l'écart-type σ en fonction de n avec trois graphes différents (séparément) : $\sigma(n)$, $\log(\sigma(\log(n))$ et $\sigma(\sqrt{n})$. Expliquer en quoi ces deux derniers plot permettent de déterminer la relation entre σ et n.

```
import numpy as np
import matplotlib.pyplot as plt

#np.random.seed(12345)
NPAS_LISTE = [50,100,200,400,800,1600,3200,6400 ]
NMARCHES = 1000

arrivee = np.empty(NMARCHES)
sigma = [] # liste vide pour stocker les valeurs de l'écart-type

def pile_face(npas):
    return ...

def marche(pas):
    ...
    return ...

for NPAS in NPAS_LISTE:
    for i in range(NMARCHES):
        arrivee[i] = marche(NPAS)

    sigma.append(np.std(arrivee))

# Affichage, ne pas oublier de décrire les axes !
plt.plot(NPAS_LISTE,sigma,'o-')
...
plt.plot(np.log(NPAS_LISTE),np.log(sigma),'o-')
...
plt.plot(np.sqrt(NPAS_LISTE),sigma,'o-')
...
```

Ce résultat pouvait être prédit théoriquement. Une analyse statistique plus complète montrerait que la position d'arrivée après n pas est une variable aléatoire qui suit une distribution de Gauss et donc que la marche aléatoire à pas discret est un modèle du processus de diffusion (voir annexe 0.3.3).

0.2.3 Marche aléatoire à pas continus en 2D

Nous avons jusqu'à présent considéré des marches aléatoires dont les pas étaient discrets et uniformément distribués (pouvant prendre la valeur ± 1 avec une probabilité égale $p = 1/2$). Bien que très intéressante pour modéliser des phénomènes physique (comme le processus de diffusion mentionné ci-dessus), il ne s'agit que d'un cas particulier. On peut aussi créer des marches aléatoires dont les pas peuvent prendre des

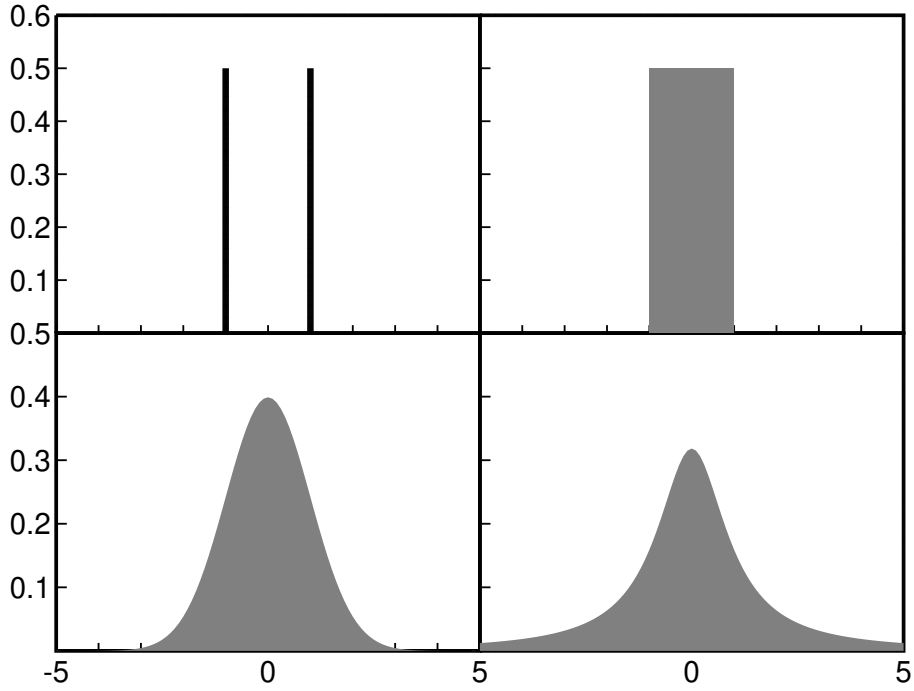


FIGURE 0.2 – Distributions pour le pas, $f(\Delta x)$. *En haut à gauche* : pas discret de ±1, *en haut à droite* : pas continue avec probabilité uniforme entre -1 et +1, *en bas à gauche* : distribution de Gauss, *en bas à droite* : distribution de Lorentz.

valeurs distribuées de façon continue sur un intervalle donné en suivant une loi de distribution donnée $f(\Delta x)$, voir figure 0.2. Ces différentes lois de probabilité sont à choisir en fonction des phénomènes physiques que l'on souhaite modéliser.

On trouve dans `numpy` des fonctions pour créer l'ensemble des distributions usuelles, en particulier la distribution de Gauss () et de Lorentz :

Gauss	$f(\Delta x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{\Delta x^2}{2}}$	`np.random.randn()`
Lorentz	$f(\Delta x) = \frac{1}{\pi} \frac{b}{(\Delta x - a)^2 + b^2}$	`np.random.standard_cauchy()`

La distribution de Lorentz (ou toute distribution "lentement" décroissante) permet de créer de façon exceptionnelle, ou rare, quelques pas "très grands" par rapport à la moyenne des autres (qu'on n'observe pas avec les autres marches aléatoires que nous avons étudiées qui décrivent des mouvements dits browniens). Elles sont aussi utiles pour modéliser d'autres phénomènes faisant intervenir des fluctuations brusques et importantes, par exemple dans le domaine de la finance. Ce type de marche est aussi utilisé pour modéliser le mouvement de prédateurs explorant un territoire à la recherches d'une proie [1] ou d'autres stratégies de recherche dans un espace donné.

Pour illustrer nous allons réaliser une marche aléatoire dans un espace bidimensionnel. Chaque pas sera déterminé par deux nombres aléatoires : un premier distribué suivant une dsitribution de Gauss ou de Lorentz pour la longueur du pas et un second distribué uniformément sur l'intervalle $[0, \pi[$ pour la direction (orientation) du pas

8. Ecrivez un programme qui simule cette marche aléatoire puis qui affiche la trajectoire $y(x)$.

1. Viswanathan, G. Fish in Lévy-flight foraging. *Nature* **465**, 1018–1019 (2010). https ://doi.org/10.1038/4651018a

```
import numpy as np
import matplotlib.pyplot as plt

#np.random.seed(12345)
NPAS = 6000
x = np.empty(NPAS)
y = np.empty(NPAS)
x[0] = 0
y[0] = 0

for i in range(1,NPAS):
    l = ...      # longueur
    theta = ... # orientation
    x[i] = ...
    y[i] = ...

plt.plot(x,y,'o-')
```

Exécutez votre programme pour des gaussiens, puis pour des pas lorentziens. Qu'observez-vous ?.

0.3 Annexes

0.3.1 Approche vectorisée

Le code ci-dessous est une vesion vectorisée de la marche aléatoire. Il présente l'avantage d'éviter la boucle for mais en contrepartie il faut stocker en mémoire tous les NPAS nombres aléatoires. L'approche la plus efficace est à apprécier selon la simulation physique envisagée. :

```
NPAS = 100

def pile_face(npas): # retourne npas pas aléatoires
    return 2 * np.random.random_integers(0,1, size=npas) -1

pas = pile_face(NPAS)
marche = np.cumsum(pas)
```

0.3.2 Calcul de l'écart-type

Pour rappel, l'écart-type σ_X d'une grandeur X est définie comme :

$$\sigma_X^2 = E\left[(X - E(X))^2\right] = \frac{1}{N}\sum_{i=0}^{N}(x_i - < X >)^2$$

où $E(X)$ est l'espérance de X, c'est-à-dire la moyenne d'une grandeur X. On pourrait utiliser cette formule pour calculer l'écart-type, mais ceci consommerait beaucoup de mémoire vive, car on devrait garder en mémoire toutes les valeurs x_i de X pour calculer la différence entre ces valeurs et $< X >$. Comme déjà pour le calcul d'une moyenne, on peut aussi éviter de garder tous les x_i en mémoire pour le calcul de l'écart-type. Pour cela, on transforme la formule de l'écart-type comme ceci :

$$\begin{aligned}\sigma_X^2 &= E\left[X^2 - 2\cdot X\cdot E(X) + (E(X))^2\right]\\ &= E(X^2) - 2\cdot E(X)\cdot E(X) + (E(X))^2\\ &= E(X^2) - (E(X))^2\end{aligned}$$

En plus de la moyenne de X, il suffit alors de calculer en même temps la moyenne de X^2, ce qui donne alors :

$$\sigma_X^2 = \frac{1}{N}\sum_{i=0}^{N} x_i^2 - \left(\frac{1}{N}\sum_{i=0}^{N} x_i\right)^2$$

Le code ci-dessous génère une série de N nombres aléatoires et calcule "à la volée" la moyenne et la variance :

```
import numpy as np

somme = 0
somme_carree = 0
N = 10000

for i in range(N):
    x = np.random.rand()
    somme += x
    somme_carree += x**2

moyenne = somme/N
variance = somme_carree/N - moyenne**2
print('moy : %.3g , var : %.3g' % (moyenne,variance))
```

0.3.3 Marche aléatoire et physique

0.3.3.1 Propriétés statistique d'une marche à pas discrets

Il est relativement simple de prédire la loi de variation de la position x_n au bout de n pas d'une marche aléatoire à pas discrets : on peut considérer que x_n est une somme de n variables aléatoires u qui chacune peuvent prendre la valeur ± 1 avec une probabilité égale $p = 1/2$. La variable aléatoire u a une valeur moyenne nulle

$$< u >= \sum_i p_i u_i = 1/2\times -1 + 1/2\times 1 = 0$$

et une variance

$$\sigma_u^2 =< u^2 > - < u >^2 = \sum_i p_i u_i^2 = 1.$$

La position moyenne $< x_n >$ après n pas peut se calculer en utilisant la linéarité de la moyenne :

$$< x_n >=< \sum_{i=1}^{n} u_i >= \sum_{i=1}^{n} < u_i >= 0.$$

De la même façon, sachant que $< x_n >= 0$ et que les u_i sont indépendants entre eux, on a pour la variance de x :

$$\sigma_n^2 =< x_n^2 > - < x_n >^2 =< (\sum_{i=1}^{n} u_i)^2 >=< \sum_{i=1}^{n} u_i^2 > + < \sum_i \sum_{j\neq i} u_i u_j >= n\sigma_u^2 = n.$$

On trouve donc que la position moyenne au bout de n pas est nulle, et que son écart-type, soit en quelque sorte la largeur typique de la distribution, vaut $\sqrt{n}$. Dans le cas de mouvements dans des espaces à deux ou trois dimensions on peut montrer qu'on obtient $\sigma = \sqrt{dn}$ où $d = 1, 2, 3$ est la dimension de l'espace.

0.3.3.2 Marches aléatoires et diffusion

Pour faire le lien avec la physique, la marche aléatoire que nous avons étudiée est en lien avec le processus de diffusion. On peut calculer par exemple comment évolue la densité $C(x, t)$ de molécules d'un certain type au cours du temps à partir d'une valeur initiale où toutes les molécules sont en $x = 0$ (soit $C(x, 0) = \delta(x)$). L'équation de la diffusion, valable à une échelle macroscopique, est donnée par :

$$\frac{dC}{dt} = D\frac{d^2C}{dx^2}$$

où la constante D est le coefficient de diffusion. Ce coefficient D est relié à la marche aléatoire à pas discrets par la relation suivante : $D = \ell^2/2\tau$ où ℓ est la longueur de chaque pas et τ est la durée moyenne entre deux pas. La solution pour une injection ponctuelle en $x = 0$ de toutes les molécules est donnée par :

$$C(x, t) = \frac{1}{\sqrt{4\pi Dt}} e^{-x^2/4Dt},$$

et est équivalente à la distribution limite que l'on obtient pour la marche aléatoire à grand nombre n de pas discrets en faisant la correspondance $t = n\tau$, $\ell = 1$ et donc $Dt = n/2$. L'écart-type de cette distribution est souvent appelée la distance caractéristique de diffusion. Quelle est l'évolution de cette distance en fonction du temps ? Comparez au cas du mouvement rectiligne uniforme.
La marche aléatoire peut être utilisée pour modéliser un grand nombre de phénomènes physiques, par exemple :

- On s'intéresse à une goutte de lait déposée dans une tasse de café. Si on connaît le temps t_0 pour que le lait se répartisse (diffuse) dans une tasse de taille donnée, combien faudrait-il pour que le lait diffuse dans une tasse 2 fois plus grande ?
- Le mouvement d'une molécule de dioxyde de carbone dans un alvéole pulmonaire est aussi un mouvement diffusif : l'alvéole peut être considéré comme une sphère de rayon $R = 100\mu m$ et le coefficient de diffusion d'une molécule de CO_2 dans l'air est $D = 0,14 cm^2 s^{-1}$. Quel est le temps caractéristique pour qu'une molécule de dioxyde de carbone se déplace de la périphérie vers le centre de l'alvéole pulmonaire ?

TP 1

Résolution numérique d'équations différentielles ordinaires - 1ère partie : la méthode d'Euler et rk4

1.1 La méthode d'Euler

Une équation différentielle du premier ordre satisfaite par une fonction $y(t)$ est de la forme,

$$\frac{dy(t)}{dt} = f\Big(y(t),\, t\Big) \tag{1.1}$$

où f est une fonction connue qui dépend du problème considéré. La méthode la plus simple pour résoudre numériquement une telle équation est la méthode d'Euler, figure 1.1 . La fonction y est calculée aux points $t_k = k\Delta t$ avec k entier, séparés par un petit intervalle Δt. La valeur de y au point $t_{k+1} = t_k + \Delta t$ est obtenue à partir de la valeur au point t_k par la formule,

$$y(t_{k+1}) = y(t_k) + f\left(y(t_k),\, t_k\right)) \times \Delta t \tag{1.2}$$

A partir de la donnée d'une condition initiale $y(t_0)$, on peut ainsi calculer de proche en proche y pour des valeurs successives de t. Les questions 1-3 proposent des applications de cette méthode à des cas très simples.

1. L'évolution dans le temps de la densité $x(t)$ d'un élément radioactif X subissant une réaction de désintégration $X \to Y$ obéit à l'équation différentielle

$$\frac{dx}{dt} = -kx \tag{1.3}$$

où k est le taux de désintégration. Écrire un petit programme qui calcule par la méthode d'Euler et enregistre x pour des valeurs de t entre 0 et $Tmax$. On prendra $k = 1$ et vous fixerez $Tmax$, Δt et $x(0)$. Tracer $x(t)$ et comparer avec la solution analytique de l'équation 1.3. La formule de la solution analytique est à trouver par soi-même sur une feuille, car c'est très simple.

Python en pratique : Interdiction d'utiliser la bibliothèque scipy ici. Pas de fonction à écrire pour l'instant. Les bibliothèques autorisées : matplotlib et numpy uniquement . On définit deux variables : la densité x et la vitesse du changement de la densité $v = -kx$, toutes les deux à l'instant t, donc juste des scalaires pas des listes ou des tableaux. De plus, on utilise des tableaux de numpy pour enregistrer dans la mémoire vive les points du temps t et la densité x. Ces tableaux sont ensuite utilisés pour faire

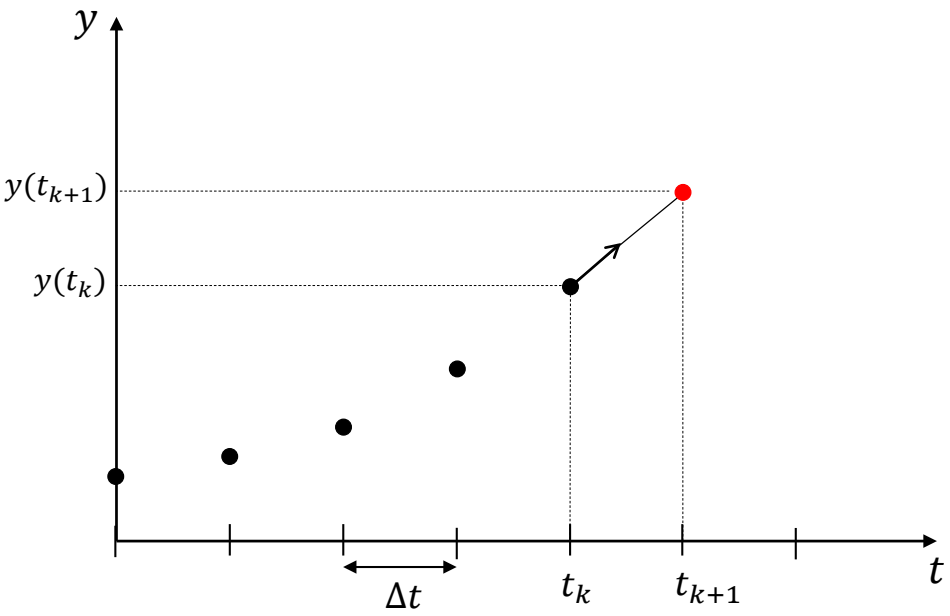


FIGURE 1.1 – Schéma méthode d'Euler.

le graphe $x(t)$ sous matplotlib. Pour une meilleure vitesse de calcul, il est toujours préférable d'utiliser des tableaux numpy à la place des listes python, car ils sont plus proches des tableaux en C, donc plus efficaces dans la gestion de la mémoire vive. Surtout, on peut ensuite accélérer le code encore plus avec la bibliothèque numba qui marche uniquement sur les tableaux numpy et pas sur les listes python. Les fonctions numpy à utiliser : empty, arange, linspace (rien d'autre ! surtout pas np.append(), voir cours-TD). Comparer les fonctions numpy : arange vs linspace. Vérifier notamment que linspace donne bien un espacement de Δt entre chaque point du temps t. Fonctions matplotlib à utiliser : plot, xlabel, ylabel, title, legend, savefig, show. Pour l'affichage de la solution analytique utiliser numpy pour la ou les fonctions nécessaires pour calculer les valeurs $x(t)$ pour tous les points du temps t utilisés pour la solution numérique. Ces valeurs ainsi que la différence par rapport à la solution numérique seront enregistrer dans deux nouveaux tableaux. Il faudra fournir deux graphes : un premier avec la superposition de la solution analytique et la solution numérique et un deuxième avec la différence des deux.

2. Supposons maintenant que l'élément Y, dont la densité est notée $y(t)$, se désintègre en un élément Z avec le taux k_2. On a alors,

$$\begin{aligned} \frac{dx}{dt} &= -kx \\ \frac{dy}{dt} &= kx - k_2 y \end{aligned} \tag{1.4}$$

Écrire un programme analogue au précédent qui calcule et enregistre $x(t)$ et $y(t)$ de $t = 0$ à $Tmax$. On prendra $k_2 = 0.1k$, $x(0) = 1$ et $y(0) = 0$. Tracer $x(t)$ et $y(t)$.

3. L'évolution d'un oscillateur harmonique est décrite par une équation du second ordre

$$\frac{d^2x}{dt^2} = -\omega_0^2 x \tag{1.5}$$

Une équation différentielle du second ordre peut être écrite sous forme de deux équations du premier ordre. Écrire ces deux équations et écrire un programme calculant

$x(t)$ avec la méthode d'Euler. Vous fixerez les paramètres et les conditions initiales. Essayer plusieurs pas de temps Δt, qu'est-ce que vous observez sur un grand nombre de périodes ?

De manière générale, une équation différentielle d'ordre supérieur à un peut être transformée en un système d'équations du premier ordre.

Le but des questions suivantes est de construire une fonction `euler` qui implémente l'algorithme d'Euler sur un pas (un seul !) et qui puisse être utilisée sans modification dans tout programme nécessitant de résoudre un système d'équations différentielles. Pour un ensemble de n équations différentielles du premier ordre satisfaites par n fonction $y_i(t)$ avec $i = 0, 1, .., n-1$,

$$\frac{dy_i(t)}{dt} = f_i\left(\{y_j(t)\}, t\right), \tag{1.6}$$

l'algorithme d'Euler pour le calcul des y_i au point $t + \Delta t$ à partir des valeurs en t peut être découpé en deux étapes :

1. le calcul de la dérivée de chaque fonction y_i au point t, donnée par $y_i'(t) = f_i\left(\{y_j(t)\}, t\right)$.
2. le calcul des y_i au point $t + dt$ avec la formule $y_i(t + \Delta t) = y_i(t) + y_i'(t) \times \Delta t$.

Les valeurs des fonctions y_i au point t sont stockées dans un tableau numpy y de taille n.

La première étape est effectuée par une fonction `deriv` :
`deriv` fournit les n dérivées des n variables stockées dans le tableau y.

- Méthode : C'est ici qu'on met la physique, i.e. les n équations différentielles du premier ordre. Soit avec une ligne de code par variable, soit avec une boucle, si c'est possible.
- Entrées :
  1. t - temps actuel
  2. y - tableau numpy de taille n pour les variables au temps t
  3. params - paramètre(s) nécessaires pour le calcul (*optionnel)*
- Sorties :
  1. dy - tableau numpy de taille n pour les dérivées au temps t

La deuxième étape est effectuée par une fonction `euler` :

But : A partir des valeurs des n variables au temps t, calculer leurs valeurs au temps t+dt (un pas de temps uniquement !) avec la méthode d'Euler

- Méthode : calcul vectoriel avec numpy, deux lignes de code, pas besoin de boucle
- Entrées :
  1. t - temps actuel
  2. dt - pas de temps
  3. y - tableau numpy de taille n pour les variables au temps t
  4. deriv - nom de la fonction qui fournit les n dérivées des n variables qui sont dans y
  5. params - paramètre(s) nécessaires pour le calcul (*optionnel)*
- Sorties :
  1. y - valeurs des n variables au temps t+dt

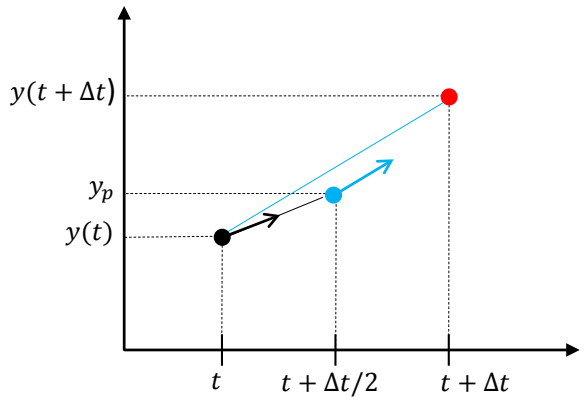


FIGURE 1.2 – Schéma méthode RK2

4. Écrire la fonction `deriv` dans le cas de l'oscillateur harmonique. Les variables simples x et v sont combinées ici dans un seul tableau numpy y de deux éléments. Ceci permet de pouvoir utiliser `euler` et `deriv` avec un système de n équations différentielles de premier ordre. La valeur du paramètre ω_0 peut être passée à la fonction via le dernier argument `params`.

5. Écrire la fonction `euler` qui reçoit en entrée les $y_i(t)$ dans le tableau y, ainsi que les valeurs de t et Δt, calcule les $y_i(t+\Delta t)$ et les stocke dans le tableau y à la place des $y_i(t)$. Il faudra dans cette fonction appeler la fonction `deriv` pour avoir le tableau avec les $y_i'(t)$. Inclure les fonctions `deriv` et `euler` dans un programme complet permettant de calculer et afficher x(t) et v(t) comme avant pour l'oscillateur harmonique.

6. Écrire un programme utilisant la fonction euler pour calculer numériquement la trajectoire dans le plan $x-y$ d'une particule de masse m et charge q dans des champs électrique et magnétique uniformes, $\overrightarrow{E} = E\,\overrightarrow{u_x}$ et $\overrightarrow{B} = B\,\overrightarrow{u}_z$. On rappelle que l'équation du mouvement est

$$m\frac{d\overrightarrow{v}}{dt} = q\left(\overrightarrow{E} + \overrightarrow{v}\times\overrightarrow{B}\right) \tag{1.7}$$

Les unités sont choisies de telle sorte que $q/m = E = B = 1$. Quel est le nombre n d'équations différentielles ? Écrire ces équations et la fonction `deriv` correspondante.

1.2 Méthode de Runge-Kutta

Pour calculer $y(t+\Delta t)$ à partir de $y(t)$, les méthodes de type Runge-Kutta estiment la valeur de la dérivée y' au milieu de l'intervalle entre les points t et $t+\Delta t$. La version la plus simple pour résoudre l'équation 1.1 procède ainsi (voir figure 1.2) :

$$\begin{aligned} d_1 &= f(y(t),t) \\ y_p &= y(t) + d_1\times\Delta t/2 \\ d_2 &= f(y_p, t+\Delta t/2) \\ y(t+\Delta t) &= y + d_2\times\Delta t \end{aligned} \tag{1.8}$$

Cette méthode s'appelle Runge-Kutta d'ordre 2 car on peut montrer que les erreurs sont d'ordre $(\Delta t)^3$.

La méthode la plus couramment utilisée est Runge-Kutta d'ordre 4 (les erreurs sont d'ordres $(\Delta t)^5$) qui utilise 4 évaluations de la dérivée. La fonction est donnée dans le fichier rk4.py (voir sur moodle) et vous pouvez l'utiliser directement en la

copiant dans votre programme. Alternativement, vous pouvez aussi l'importer avec `from rk4 import rk4`, il faut juste mettre le fichier rk4.py dans le même dossier que votre code.

7. Résoudre l'équation $\frac{d^2x}{dt^2} = -\omega_0^2 x$ en utilisant la fonction `euler` avec $\Delta t = 0.01s$. Tracer $x(t) - x_{th}(t)$ où $x(t)$ est la solution numérique et $x_{th}(t)$ est la solution analytique exacte. Que constatez-vous ? Diminuer le Δt. Le problème est-il résolu ?

8. Refaire la question précédente en utilisant à présent la fonction fournie `rk4`. Pour cela il suffit d'appeler la fonction `rk4` au lieu d'appeler la fonction `euler`. Que constatez-vous ?

1.3 Réduire le temps de calcul avec numba

Python en tant que langage interprété n'est pas très rapide pour exécuter des boucles, car chaque ligne de code doit d'abord être interprétée (=traduit en langage machine) avant d'être exécutée, même quand c'est toujours la même ligne de code dans une boucle. Il existe des langages compilés qui n'ont pas ce défaut, comme le C/C++ ou le Fortran, ou encore des langages qui ont un compilateur à la volée ("just-in-time" (jit) compiler en anglais) comme le nouveau langage Julia, qui ressemble beaucoup au python tout en s'éxecutant à la vitesse du C.

Mais il existe heureusement une bibliothèque en python qui ajoute cette fonctionnalité du jit-compiler : numba (www.numba.org)

Numba s'utilise très facilement, tant qu'on respecte certaines règles :

1. Utiliser uniquement des tableaux numpy et non des listes ou autres objets python, qui sont plus difficile à traduire par numba.
2. Ne pas utiliser des variables globales pour les fonctions accélérées par numba, à moins qu'il ne s'agisse de constantes qui ne changent pas lors de l'exécution du programme. Numba considère les variables globales comme des constantes, même si on change la valeur d'une variable globale dans une fonction ou le code principal hors numba. Si on souhaite par exemple exécuter une fonction accélérée par numba plusieurs fois et changer une variable globale entre chaque exécution (ex : paramètre physique comme la température), cela ne marche pas, numba conserve la valeur que cette variable avait lors du premier appel à la fonction (= le moment où la fonction est compilée par numba). Il faut mettre alors cette variable en tant qu'argument dans la fonction.
3. Ne pas utiliser `dtype=float` ou `dtype=int` lors de la création d'un tableau numpy, car les types `float` ou `int` sont des types de base de python et non de numpy, donc pas supportés par numba ! Il faut utiliser un type de numpy, comme : `dtype=np.float64` ou `dtype=np.int32`.

Il suffit ensuite d'importer numba avec

```
from numba import jit
```

puis d'ajouter l'instruction

```
@jit(nopython=True)
```

avant chaque déclaration de fonction qui doit être accélérée par numba. Alternativement on peut mettre aussi

```
from numba import njit
```

avec le raccourci `@njit` avant chaque déclaration de fonction, comme ceci :

```
from numba import njit

@njit
def fonction(x, y):
```

Il faut faire attention de mettre cette instruction (`@njit` ou `@jit(nopython=True)`) aussi sur toutes les sous-fonctions qu'on a écrites soi-même et qu'une fonction accélérée par numba pourrait appeler.

9. Enregistrer votre programme précédent qui utilise `rk4` sous un nouveau nom et modifier le pour pouvoir utiliser numba. Pour cela il faut mettre la boucle sur le temps qui appelle `rk4` dans une nouvelle fonction, afin de pouvoir accélérer cette fonction avec numba. Toutes les sous-fonctions qu'appelle cette fonction doivent également être accélérées avec numba, i.e. on doit ajouter le raccourci `@njit` devant cette nouvelle fonction et devant les fonctions `rk4` et `deriv`. `rk4` doit alors être copié directement dans votre code. Comparer le temps d'exécution de votre programme accéléré par numba avec celui de votre programme sans numba. Pour cela vous pouvez utiliser la bibliothèque `time` de python : il suffit d'enregistrer le temps au démarrage et à la fin avec :

```
start = time.time()

end = time.time()
```

Choisir un nombre d'itérations suffisamment grand pour pouvoir mesurer des différences entre les deux versions de votre code (avec et sans numba). Astuce : pour convertir un code optimisé avec numba en code standard sans numba, il suffit simplement de commenter la ligne avec `@njit` => `#@njit`

1.4 Remarque : Avantages de rk4 par rapport aux fonctions odeint et solve_ivp de scipy

Cette sous-section ne contient pas de question et n'est pas un prérequis pour la suite, il s'agit juste de mieux comprendre pourquoi l'utilisation de rk4 peut être avantageuse par rapport aux fonctions déjà fournies par scipy. Donc si vous n'avez plus le temps, vous pouvez lire cette partie plus tard.

Alternativement à la fonction rk4, on peut aussi utiliser odeint (obsolète normalement) ou solve_ivp (nouvelle fonction qui remplace odeint) de scipy. Voici un example d'utilisation des deux fonctions :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy.integrate import solve_ivp

def deriv1(y,t,omega):
    dy = np.zeros(2)
    dy[0] = y[1]
    dy[1] = -omega**2 * y[0]
    return dy
```

```python
# argument order is different for solve_ivp()
def deriv2(t,y,omega):
    dy = np.zeros(2)
    dy[0] = y[1]
    dy[1] = -omega**2 * y[0]
    return dy

t_max = 10
N_pas = 100
time_vec = np.linspace(0, t_max, N_pas)
yinit = (1,0)
omega = 1
result = odeint(deriv1, yinit, time_vec, args=(omega,))
plt.plot(time_vec, result, '-o')
plt.show()
result2 = solve_ivp(deriv2, (0,t_max), yinit, args=(omega,),
↪  t_eval=time_vec, rtol=1e-9)
plt.plot(result2.t, result2.y[0], '-o')
plt.plot(result2.t, result2.y[1], '-o')
plt.show()
```

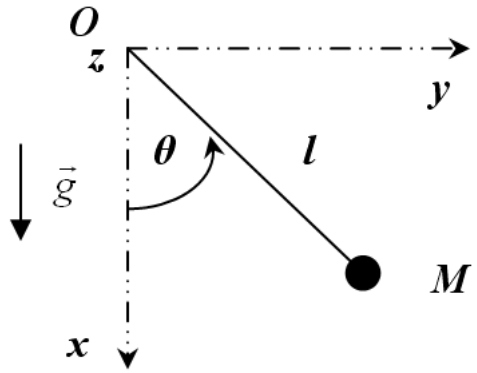
Comme ces deux fonctions sont des boites noires et sont relativement difficiles à utiliser, car déjà très avancées (surtout solve_ivp), nous recommandons d'utiliser plutôt la fonction rk4 fournie ici, à moins que vous ayez des cas très difficiles pour l'algorithme RK4 (ce qui est relativement rare) qui auraient besoin d'un pas de temps adaptatif, comme c'est implémenté dans solve_ivp.

La fonction rk4 a aussi d'autres avantages du fait qu'elle ne fait qu'un seul pas de temps et non tous les pas pour un temps de simulation donné :

1. Comme pour odeint ou solve_ivp, la fonction rk4 n'a pas besoin d'être modifiée en fonction du problème de physique, ce qui évite d'y introduire des erreurs, surtout si on l'importe avec `from rk4 import rk4`
2. On peut choisir combien de points sont enregistrés dans la mémoire, car souvent on essaye d'avoir un pas de temps très fin, sans pour autant vouloir produire des millions points (bonjour les plots !). C'est possible aussi avec solve_ivp via t_eval.
3. Si jamais on souhaite arrêter l'intégration numérique avant t=tmax, par exemple parce que le vaisseau spatial est tombé sur la terre, c'est faisable avec un simple test if dans la boucle sur le temps. Autre cas de figure : le pendule où on limite l'angle entre -180 et +180 avec des tests if dans la boucle sur le temps, même si ici on pourrait filtrer les résultats en "post-processing" après la boucle sur le temps. L'exemple du diagramme de bifurcation obtenu à partir du pendule chaotique montre une autre application pour ne pas enregistrer tous les points et de faire la correction -180/180 en même temps.
4. Cela permet aussi d'échanger facilement l'algorithme, il suffit simplement de changer le nom de la fonction dans le programme sans toucher au reste.
5. La fonction rk4 a montré sa robustesse pendant des années en L3 physique numérique à Sorbonne Université dans sa version en C. Le seul hic sous python est la performance qu'il faut rattraper via numba pour s'approcher des performances de odeint ou solve_ivp.

TP 2

Résolution numérique d'équations différentielles ordinaires - 2ème partie : application au pendule chaotique



2.1 Étude du mouvement d'un pendule avec l'approximation des petits angles

On considère le pendule simple de la figure précédente, dont l'équation du mouvement libre s'écrit :

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \Omega^2 \sin\theta = 0 \qquad \text{avec } \sin\theta \simeq \theta \quad \rightarrow \quad \frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \Omega^2\theta = 0$$

où θ est l'angle que fait le pendule par rapport à la verticale, $\Omega = \sqrt{g/l}$ est la pulsation propre et q est le terme de frottement fluide. On utilisera par commodité la valeur suivante : $\Omega = 1\text{rad.s}^{-1}$.

1. Résolvez cette équation linéarisée ($\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \Omega^2\theta = 0$) avec la méthode RK4 pour différentes valeurs de l'amortissement : $q = 1$, $q = 2$, $q = 5\ s^{-1}$ et tracez *sur un même graphe* l'évolution de $\theta(t)$ dans ces régimes respectivement pseudo-périodique, critique et apériodique. On prendra comme conditions initiales $\theta(t = 0) = 10°$ (à convertir en radian) et $\frac{d\theta}{dt}(t = 0) = 0$ et un pas de temps $dt = 0.05$s pour t allant de 0 à 20s. Il est utile ici de mettre la boucle sur le temps qui calcule $\theta(t)$ dans une fonction avec comme paramètre la valeur de l'amortissement q. Cette fonction devra renvoyer deux tableaux numpy : celui qui contient les valeurs du temps t et celui qui contient les valeurs de $\theta(t)$.

On ajoute maintenant une force d'excitation au pendule de sorte que l'équation du mouvement s'écrive :

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \Omega^2\theta = F_e \sin(\Omega_e t).$$

2. Résolvez cette nouvelle équation avec la méthode RK4 pour une force excitatrice d'intensité $F_e = 1\text{rad.s}^{-2}$ et de pulsation $\Omega_e = 2\Omega/3$. Tracez *sur un même graphe* la trajectoire dans l'espace des phase $(\theta, \frac{d\theta}{dt})$ pour le pendule libre ($q = 0$ et $F_e = 0$) , amorti (q=1 et $F_e = 0$), et amorti avec excitation (q=1 et $F_e = 1$). On prendra toujours comme conditions initiales $\theta(t = 0) = 10°$ (à convertir en radian) et $\frac{d\theta}{dt}(t = 0) = 0$. Commentez la forme des trajectoires que vous observez.

2.2 Mouvement chaotique

Lorsque l'on ne fait plus l'hypothèse des petits angles ($\sin\theta \simeq \theta$), on obtient une équation différentielle d'ordre 2 qui n'est pas linéaire :

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \Omega^2 \sin\theta = F_e \sin(\Omega_e t).$$

Pour certaines valeurs des paramètres physiques, le comportement du pendule sera chaotique. Afin d'illustrer ce comportement, on se placera dans les conditions suivantes : $\theta(t = 0) = 10°$ (à convertir en radian par votre script python) et $\frac{d\theta}{dt}(t = 0) = 0$, $\Omega_e = 2\Omega/3$, $q = 0.5s^{-1}$.

3. Résolvez l'équation du mouvement non-linéaire avec la méthode RK4 pour les valeurs suivantes de l'amplitude d'excitation $F_e = \{1.4, 1.44, 1.465, 1.5\}$ rad.s^{-2} et tracez $\theta(t)$ sur un temps de 100s (choisir le nombre de plots nécessaires pour bien distinguer le comportement). Ajouter deux tests **if** dans la boucle après l'appel à rk4 pour maintenir l'angle θ dans l'intervalle $[-\pi; \pi]$. Que constatez-vous au sujet de la période du pendule ? (attention, périodique≠sinusoïdal...)

4. Dans le cas $F_e = 1.5$ rad.s^{-2}, calculez l'évolution de $\theta(t)$ pour deux conditions initiales très proches l'une de l'autre : $\theta(t = 0) = 10°$ et $\theta(t = 0) = 9.999°$. Tracez les deux trajectoires sur un même plot. Au bout de combien temps est-ce qu'un écart entre les deux courbes devient visible ? On voudrait maintenant quantifier comment et avec quelle la vitesse cet écart croit. On essaye de vérifier une croissance exponentielle, c'est-à-dire la valeur absolue de cet écart croit comme $e^{\lambda t}$ où λ est « l'exposant de Lyapounov » qui caractérise la vitesse à laquelle deux systèmes quasiment identiques divergent. Essayer de déterminer l'exposant de Lyapounov en traçant le logarithme de cet écart absolu :

```
plt.plot(tValues, np.log(np.abs(theta2 - theta1)))
```

Puis en y superposant une droite à la main, vérifiant ainsi qu'il s'agit d'une croissance exponentielle. La pente de cette droite est la valeur de l'exposant de Lyapounov λ. Qu'en conclure sur vos capacités de prédiction de l'état futur du système ?

2.3 Diagramme de bifurcation.

Afin de mettre en évidence la transition du système vers le régime chaotique, on peut calculer l'état du système uniquement à des instants t_n tels que $t_n = \frac{2\pi n}{\Omega_e}$ où n est un entier, ce qui veut dire qu'on l'observe en phase avec la fréquence excitatrice. Comme dans le cas où on observe un mouvement avec un stroboscope, si le pendule est dans un

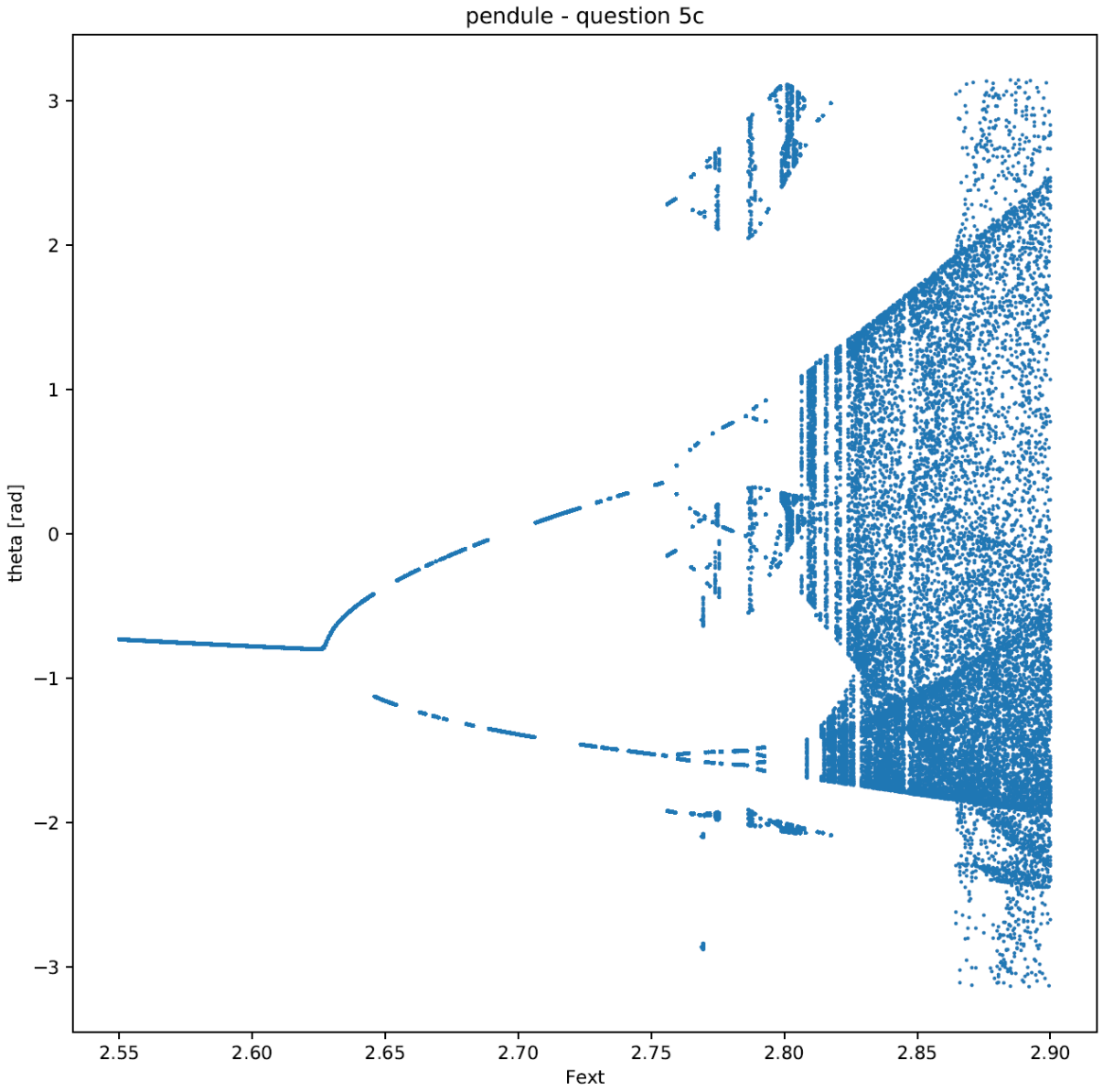


FIGURE 2.1 – Diagramme de bifurcation

régime stationnaire et oscille à la fréquence d'excitation, alors les valeurs de θ_n que l'on calcule ont la même valeur, c'est comme si on avait figé le mouvement. En faisant varier la valeur de la force d'excitation, vous pourrez étudier les changements de régime du pendule.

5. Pour tracer le diagramme de bifurcation, il faut faire évoluer le système suffisamment longtemps pour éliminer le régime transitoire, et afficher par exemple 50 ou 100 valeurs de θ pour des temps multiples entiers de la période d'excitation. On choisira un δt adapté pour faciliter les calculs, par exemple :$\delta t = \frac{2\pi}{200\Omega_e}$. Pour obtenir de beaux exemples, vous pourrez faire varier F_e entre 1.41 et 1.5 rad.s^{-2} par pas de 0.005, ou entre 2.55 et 2.90. Vous pourrez obtenir un graphe comme montré dans la figure 2.1.

TP 3

Adsorption de particules sur une surface

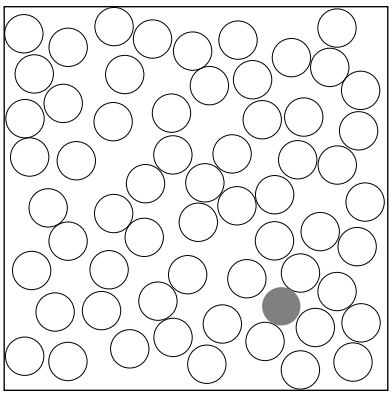
3.1 Surface homogène

L'exposition d'une surface cristalline à un gaz donne lieu à des *phénomènes d'adsorption* : les particules du gaz sont piégées sur la surface du cristal. Ce phénomène a de nombreuses applications, en particulier dans la réalisation de catalyseurs.

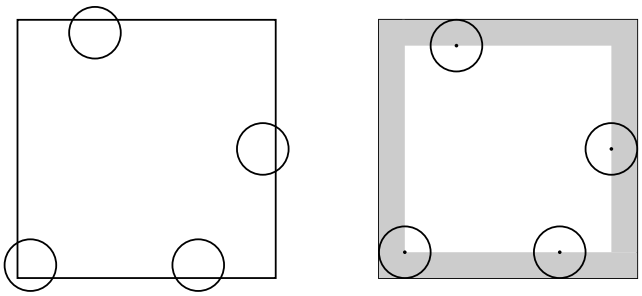
Pour modéliser ce problème, on fait les hypothèses préliminaires suivantes :

- La surface cristalline est un carré de côté L.
- Les particules de gaz adsorbées sont modélisées par des disques de rayon R avec $R \ll L$. Ces disques sont impénétrables, c'est-à-dire que deux particules ne peuvent pas se chevaucher. (Voir la figure 3.1a.)
- Une fois qu'une molécule a été adsorbée, elle ne bouge plus et ne quitte plus la surface du cristal.

La simulation fonctionne ainsi : on part d'une surface vide et, à chaque pas de temps, on essaye de rajouter une particule de gaz. La nouvelle particule arrive à un endroit aléatoire si elle ne chevauche aucune particule déjà présente, on la garde, sinon rien



(a) Un exemple de particules sphériques adsorbées aléatoirement sur une surface. La particule indiquée en gris est la dernière que l'on a pu caser.



(b) À gauche : configuration interdite car les particules débordent. À droite : configuration valide. Les particules sont à l'extrême limite de la zone autorisée.

FIGURE 3.1 – Surface homogène

ne se passe et le système n'est pas modifié. On fait ainsi de nombreux essais jusqu'à ce que l'on n'arrive plus à caser de nouvelles particules et on s'intéresse aux propriétés de l'état final, en particulier au nombre de particules que l'on a réussi à placer.

Une particule de gaz adsorbée doit-être entièrement contenue dans le carré (voir figure 3.1b). Si on note (x, y) les coordonnées du centre de la particule, les valeurs autorisées pour x et y en fonction de L et de R sont comprises dans l'intervalle $[R, L-R]$.

Deux particules de coordonnées (x, y) et (x', y') se chevauchent si leur distance est inférieure à $2R$, c'est à dire :

$$\sqrt{(x-x')^2 + (y-y')^2} < 2R$$

On peut donner une *borne supérieure* au nombre de particules de rayon R que l'on peut espérer caser sans chevauchement dans un carré de côté L, en prenant le rapport entre la surface du carré et la surface d'un cercle :

$$\text{N\_MAX} = \frac{L^2}{\pi R^2}$$

Sans chevauchement cette borne ne sera évidemment jamais atteint, car il restera toujours des surfaces non-couvertes par des cercles. `N_MAX` sera utile pour réserver à l'avance un nombre suffisant de cases dans un tableau qui contiendra les coordonnées des cercles.

1. Écrire un programme capable de simuler ce phénomène d'adsorption. Le programme doit essayer de placer successivement des particules dans le système jusqu'à ce qu'il y ait eu `MAX_TRIES` échecs consécutifs, c'est-à-dire qu'après `MAX_TRIES` d'essais pour placer la dernière particule, il n'a trouvé aucun emplacement libre.

A la fin le programme devra afficher le nombre de particules placées ainsi que le rapport entre la surface totale occupée par toutes les particules adsorbées et la surface du carré. Les constantes L et R et `MAX_TRIES` seront des constantes et l'on pourra prendre, par exemple

```
L = 20.0
R = 0.4
MAX_TRIES = 1000
```

3.1.1 Conseils pour la mise en place du programme

Pour réaliser votre programme de la question précédente, voici un certain nombre de conseils sur les ingrédients à mettre en place. Ces conseils sont là pour vous aider à une mise-en-place rapide et robuste par rapport aux "bugs", mais vous êtes libre de les suivre ou pas et de faire vos propres choix des éléments de langage du python qui s'offrent ici.

3.1.1.1 La structure de données pour représenter les cercles

Comme tous les cercles ont le même rayon R, la seule chose qu'il faut enregistrer dans la mémoire vive sont les coordonnées (x, y) des centres des cercles. Pour cela on utilise ici deux tableaux numpy, un pour x et un pour y.

Lors de l'adsorption les cercles seront ajoutés successivement à la surface et devront alors également être enregistrés dans la mémoire vive successivement. Comme le processus est aléatoire, il est impossible de savoir l'avance combien de cercles seront

enregistrés à la fin d'un remplissage du carré. On sait par contre que ce nombre n'atteindra jamais `N_MAX` (voir plus haut). La structure de données adaptée ici est un "tableau dynamique", c'est à dire un tableau qui peut changer de taille, comme ici croître d'une case avec chaque ajout de cercle.

Ici le tableau dynamique est implémenté avec un tableau statique de la bibliothèque numpy de taille `N_MAX` et grâce à l'utilisation d'une variable `n` qui représente le nombre de cercles déjà présents dans le système. Attention la taille d'un tableau est un entier, donc aussi `N_MAX` doit être un entier.

On définira, dans la fonction gérant l'évolution de notre système, trois variables pour indiquer le nombre et la position des cercles adsorbées :

```
n = 0       # nombre de cercles déjà présents dans le système
x = np.empty(N_MAX)    # x[i] est l'abscisse du i-ème cercle présent
y = np.empty(N_MAX)    # y[i] est l'ordonnée du i-ème cercle présent
```

Ces variables seront ensuite transmises par argument aux différentes fonctions qui les utilisent et/ou les mettent à jour. Il faut noter que `x[i]` et `y[i]` ne sont définis que pour $0 \leq i < n$, donc pour parcourir les coordonnées des cercles déjà adsorbés, on fait :

```
for i in range(n):
    print(x[i], " ", y[i])
```

Au départ on a `n = 0`, c'est cette variable qui détermine l'emplacement des deux tableaux où on doit ajouter un nouveau cercle : x[n] = x_new. Après chaque ajout `n` est incrémenté de un. Par construction `n` doit toujours rester bien inférieur à `N_MAX`, sinon il s'agit d'un bug. Pour vider la surface il suffit de faire `n=0`, pas besoin d'initialiser les valeurs des tableaux `x` et `y` ici.

3.1.1.2 L'organisation du code

Une bonne manière d'écrire le programme est d'utiliser les fonctions suivantes :

- Une fonction coord(L, R) qui renvoie une valeur aléatoire dans l'intervalle autorisé pour la coordonnée x ou y de la nouvelle particule qu'on essaye de placer. On n'a besoin d'écrire qu'une seule fonction coord(L, R) pour les deux coordonnées x et y, comme il s'agit d'une surface carrée. Pour obtenir les deux nouvelles coordonnées `x_new` et `y_new`, on appellera coord(L, R) deux fois. On n'utilise pas ici rng.random() de numpy, mais np.random.rand() pour générer un nombre aléatoire réel entre 0 et 1. Cela évite de devoir passer un objet rng à travers les fonctions, mais aussi surtout d'être compatible avec numba, car rng.random() ne fonctionne pas avec numba.
- Une fonction place_libre(n, x, y, x_new, y_new) qui vérifie si pour les nouvelles coordonnées `x_new` et `y_new` la place est libre, i.e. qu'il n'y a pas de chevauchement avec les particules déjà placées. Cette fonction retourne 0 si la place n'est pas libre et 1 si la place est libre. Quand il n'y a pas encore de particules adsorbées, `n=0`, la fonction doit bien sûr retourner 1. Sinon pour `n>0` elle doit calculer dans une boucle toutes les distances entre le centre `(x_new, y_new)` de la nouvelle particule et les centres des particules déjà adsorbés. Dès qu'elle trouve une distance inférieure à $2R$, alors elle peut arrêter la boucle avec `break`.
- Une fonction remplissage(L, R, MAX_TRIES) qui gère l'évolution du système pour arriver à un remplissage de la surface, jusqu'à on a `MAX_TRIES` échecs consécutifs. Elle renvoie le nombre de cercles adsorbés.

La fonction remplissage doit effectuer plusieurs tâches :

```
def remplissage(L, R, MAX_TRIES):
    ...  # créer tableaux numpy x et y pour coordonnées des cercles
    n = 0
    echecs = 0
    while echecs < MAX_TRIES:
        # tirage aléatoire de coordonnées pour placer un nouveau cercle
        x_new = coord(L, R)
        y_new = coord(L, R)
        # Test si l'emplacement est libre:
        libre = place_libre(n, x, y, x_new, y_new)
        # Ajout de cercle si emplacement libre
        if libre == 1:
            ... # ajout du nouveau cercle dans les tableaux numpy x et y
            echecs = 0  # remettre le compteur à zéro pour le prochain
            ↪ cercle
        else:
            echecs += 1  #incrémenter échecs
    # fin boucle while
    return x[:n], y[:n], n  # retourner la partie des tableaux x et y
    ↪ qui a été remplie et le nombre de cercles adsorbés
```

3.1.2 Analyse des résultats

2. Appeler votre fonction `remplissage` avec les paramètres donnés plus haut et visualiser la configuration finale. Pour cela vous pouvez utiliser ce code [1] utilisant matplotlib :

```
import matplotlib.pylab as plt
import matplotlib # pour collections
# crée une liste d'objets "Circle" de matplotlib avec les coordonnées
↪ extraites des tableaux x et y :
circles = [plt.Circle((xi,yi), radius=R, linewidth=0, color='b') for
↪ xi,yi in zip(x,y)]
# crée une "collection" des cercles pour matplotlib, afin de tracer tous
↪ les cercles d'un seul coup (c'est mieux au niveau temps de calcul):
c = matplotlib.collections.PatchCollection(circles)

plt.scatter(x,y,s=1) # nécessaire pour avoir les bons axes
plt.axis('scaled')  # pour un avoir un carré et pas un rectangle
plt.xlim(0, L)
plt.ylim(0, L)
plt.gca().add_collection(c) # tracer la collection de cercles

plt.savefig("graph02.pdf", bbox_inches='tight')
plt.show()
```

Vérifier bien qu'il n'y a pas de chevauchement entre les cercles et qu'aucun cercle dépasse les bords du carré. Une fois que vous avez ainsi graphiquement vérifié que

1. trouvé en partie ici : `https://stackoverflow.com/questions/48172928/`

votre programme fonctionne bien, enlever pour la suite (dans un nouveau fichier `prog03.py`) les lignes de code qui servent à afficher les cercles.

3. Modifier votre programme pour que l'expérience soit faite `M` = 20 fois, afin de calculer la moyenne sur toutes ces expériences du nombre de particules adsorbées et de fraction de la surface du carré qu'elles occupent. Ici bien sûr on ne visualise pas les configurations obtenues. Calculer aussi l'écart-type de ces deux variables mesurées. Vous pouvez utiliser ici les fonctions numpy `np.mean()` et `np.std()`. Comme ici on a deux grandeurs A et B (ici n et η) qui sont reliées simplement par un facteur d'échelle, il suffit de faire le calcul de l'écart-type pour une seule grandeur, puis d'appliquer ce facteur d'échelle pour obtenir l'écart-type de l'autre grandeur :
Supposons que $b_i = k \cdot a_i$, alors on obtient par simple insertion dans la formule de l'écart-type :

$$\sigma_B^2 = k^2 \cdot \sigma_A^2$$

Pour vérifier vos résultats, voici les valeurs approximatives qu'on devrait obtenir :

- La moyenne du nombre de particules adsorbées : $< \mathrm{n} >= 373 \pm 8,4$
- La moyenne de la fraction de surface : $\eta =< \mathrm{n} > \cdot \pi R^2/L^2 = 0,47 \pm 0,01$

4. Avec une simple comparaison de valeurs sans faire un nouveau script python : Comment se comparent les valeurs obtenues à la fraction qu'on pourrait idéalement occuper d'une façon ordonnée ?

Voici deux exemples d'empilement ordonné : l'empilement carré et hexagonal (voir figure 3.2). Pour l'empilement carré d'une surface avec $L = n \cdot 2R$ la fraction de surface occupée est simplement :

$$\eta = \frac{\pi R^2}{4R^2} = \frac{\pi}{4} = 0,785$$

L'empilement hexagonal est l'empilement le plus compact possible pour des cercles de même taille, ce qui a été démontré par Joseph Louis Lagrange en 1773. La fraction de surface occupée se calcule par exemple comme ceci :

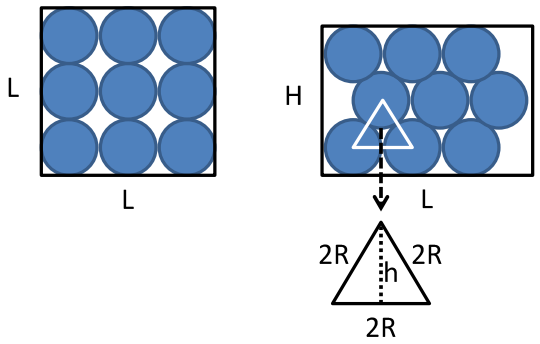


FIGURE 3.2 – Empilement ordonné, *à gauche :* empilement carré, *à droite :* empilement hexagonal

On suppose une surface rectangulaire $L\,x\,H$, avec $L = n \cdot 2R$ et m étant le nombre de lignes de cercles. Le nombre de cercles pour une surface infinie ($L \to \infty$) est alors égale à $m \cdot n$. La distance entre deux lignes est égale à $h = \sqrt{3}R = 1,73R$, comme les centres de trois cercles sont reliés par un triangle équilatéral tel qu'on voit dans la figure 3.2, ce qui donne la relation $R^2 + h^2 = 4R^2$. Donc la hauteur H de la surface peut être exprimé comme ceci, si on suppose un très grand nombre de lignes m :

$$H = 2R + (m-1)h \approx m \cdot h = m\sqrt{3}R$$

Au final on obtient pour l'empilement hexagonal d'une surface $L\,x\,H$ très grande par rapport à R :

$$\eta = \frac{m \cdot n \cdot \pi R^2}{L \cdot H} = \frac{m \cdot n \cdot \pi R^2}{n \cdot 2R \cdot m\sqrt{3}R} = \frac{\pi}{2\sqrt{3}} = \frac{1}{6}\pi\sqrt{3} = 0,907$$

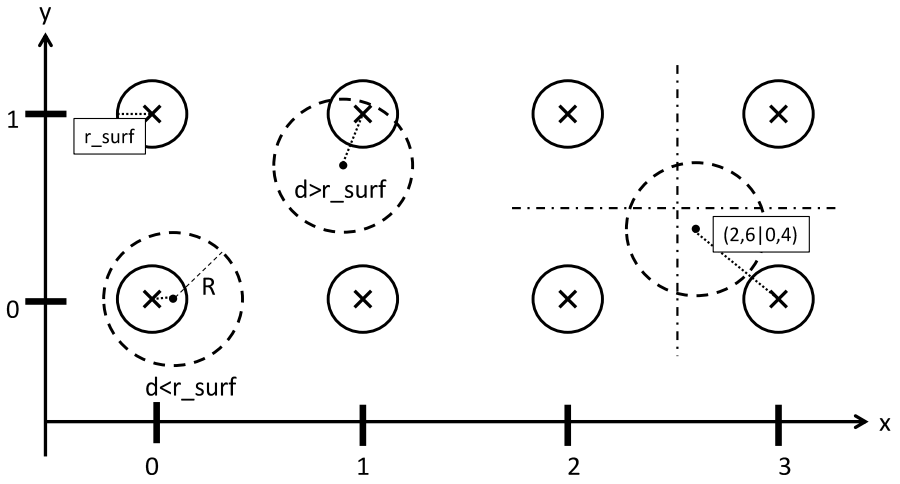


FIGURE 3.4 – Modèle d'adsorption sur un réseau d'atomes, ici en cercles pleins avec croix. Les particules de gaz, ici en cercles avec tirets, peuvent soit s'adsorber dans tous les cas, si $d <$ r_surf ou alors seulement avec une certaine probabilité, si $d >$ r_surf. Comme les atomes ont des coordonnées entières (0,1,2,...), il est très facile de trouver l'atome le plus proche à partir des coordonnées de la particule de gaz : Il suffit de les arrondir vers l'entier le plus proche, ici par exemple : (2,6|0,4) => (3|0).

3.2 Surface structurée

On a jusque-là modélisé ce problème en considérant la surface d'adsorption comme parfaitement homogène. En fait, il y a des interactions entre les atomes de la surface d'adsorption et les particules du gaz, et il se trouve que les particules du gaz s'adsorbent plus facilement à proximité des atomes de la surface d'adsorption.

Pour prendre le cas le plus simple, on suppose que les atomes de la surface d'adoption sont organisés selon un réseau cristallin carré de côté 1 (voir figure 3.3) : il y a un atome de la surface en tous les points de coordonnées (x, y) où x et y sont des entiers compris entre 0 et L.

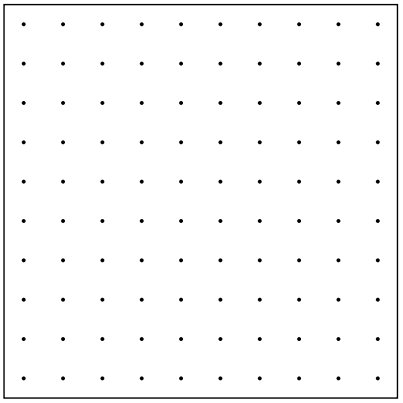


FIGURE 3.3 – Surface d'adsorption avec sa structure atomique, les atomes étant organisés en réseau carré.

Décrire précisément ces interactions est difficile, mais on obtient de bons résultats avec le modèle simplifié suivant (voir figure 3.4) :

- On suppose qu'il existe une distance caractéristique `r_surf` qui décrit la portée de l'interaction entre les atomes de la surface et les particules du gaz.
- Si une particule du gaz essaye de s'adsorber à une distance inférieure à `r_surf` de l'un des atomes de la surface, l'adsorption a toujours lieu.
- Si, au contraire, une particule du gaz essaye de s'adsorber à une distance supérieure à `r_surf` de tous les atomes de la surface, l'adsorption n'a lieu qu'avec une probabilité $\exp(-U/T)$ où $U > 0$ représente le "coût" énergétique à se placer loin

des particules de la surface et où T représente la température. (Pour être parfaitement précis, T est en fait la température en Kelvin multipliée par la constante de Boltzmann k_B.)

(Le modèle qu'on vient de décrire est une variante de "l'algorithme de Métropolis". L'algorithme de Métropolis est très répandu en physique et ailleurs. Voir l'article original de Nicholas Metropolis et al. (1953) sur moodle.)

5. Pour quelle valeur de T retrouve-t-on le modèle de la première partie ? Comment pourrait-on décrire en quelques mots le modèle obtenu en prenant $T = 0$?

3.2.1 Mise en place du programme

6. Modifier votre programme pour prendre en compte les atomes de la surface et la température. La distance `r_surf` et l'énergie U seront définies par des constantes et l'on pourra prendre, par exemple,

```
r_surf = 0.05
U = 10.0
```

Il faudra

- Définir une fonction dist_latt (x_new, y_new) qui renvoie la distance du centre de la particule de gaz de coordonnées `(x_new, y_new)` à l'atome le plus proche de la surface d'adsorption. On pourra utiliser la fonction np.rint de numpy qui renvoie l'entier le plus proche de son argument (voir figure 3.4 pourquoi cela est utile).
- Modifier la fonction remplissage. Cette fonction prend désormais en argument la température en plus : **int** remplissage (..., T) et doit implémenter l'algorithme décrit.
- Modifier votre programme pour simuler le modèle à une température donnée (par exemple, $T = 0$ avec `MAX_TRIES = 10000`).

Quelques aides :

- Pour accepter l'adsorption seulement avec la probabilité $p = \exp(-U/T)$ on pourra utiliser `np.random.rand()` qui fournit un nombre (pseudo-)aléatoire entre zéro et un d'une manière uniforme. La probabilité que la condition `np.random.rand() < p` est vraie est alors égale à p, si $p \in [0, 1]$. Comme $U > 0$ et $T \geq 0$, on a bien $\exp(-U/T) \in [0, 1]$ et on peut alors utiliser `np.random.rand() < exp(-U/T)` comme condition pour accepter l'adsorption pour les cas $d >$ r_surf.
- Pour pouvoir inclure ici le cas $T = 0$, il faut transformer cette condition en : `T*log(np.random.rand()) < -U`

3.2.2 Dépendance de la température

7. Comme avant utiliser matplotlib pour visualiser séparément les configurations finales obtenues pour ces températures T : 0, 1, 2, 5 et 10, (voir au début de la section 3.1.2 avec `M=1`). Décrivez qualitativement ce que vous observez. Surtout pour des températures proche de zéro il faut utiliser `MAX_TRIES = 10000` pour éviter d'avoir trop de trous à cause d'une recherche pas assez profonde.

8. Modifier votre programme pour faire `M=100` simulations indépendantes pour chaque température entre 0 et 10 d'un pas de $\Delta T = 0,5$, et tracer le graphe de la fraction moyenne de la surface occupée en fonction de la température. Utiliser ici un `MAX_TRIES = 10000`. Vue la lenteur du calcul avec python de base, il est nécessaire d'utiliser numba ici, voir la section 3.4.1.

9. En ayant analysé les configurations avec matplotlib (question plus haut), comment peut-on expliquer la courbe de la question précédente ? Peut-on définir une "température critique" ? Avez vous une prédiction théorique pour la valeur de la fraction à basse température ?

3.3 Pour aller plus loin : MAX_TRIES et numba

Revenons sur la partie avec une surface homogène :

10. Répéter pour plusieurs valeurs de `MAX_TRIES` le calcul de la fraction moyenne (taux) de surface occupée par les atomes adsorbés ainsi que l'écart-type. Augmenter `MAX_TRIES` avec un facteur deux entre chaque simulation, comme ceci : 1000, 2000, 4000, ..., 32000. Utiliser `M` = 20 répétitions seulement pour limiter le temps de calcul. Tracer la fraction moyenne de surface occupée en fonction de MAX_TRIES en échelle semi-log. Pour afficher également l'écart-type sous forme de barres d'erreurs, vous pouvez utiliser `errorbar` de matplotlib à la place de la commande `plot` :

```
plt.errorbar(max_tries, moy_fracSurface, yerr = ecart_fracSurface)
```

Commentez le graphe.

11. Accélérer l'exécution de votre code avec numba, voir section 3.4.1. Essayer d'étendre la courbe jusqu'à `MAX_TRIES` = 512.000 et avec `M` = 100 répétitions. Commentez le nouveau graphe.

3.4 Annexe

3.4.1 numba

Pour que le temps de calcul ne rend pas impossible la réalisation de certains exercices, l'utilisation de numba devient incontournable ici, car on peut obtenir avec numba une accélération de trois ordres de grandeurs ici. Pour pouvoir utiliser numba, il faut utiliser np.random.rand() et non rng.random() pour générer un nombre aléatoire réel entre 0 et 1, comme indiqué au début de ce TP. De plus il faut faire attention que `N_MAX` est un entier, car comme le python de base, numba n'accepte pas `np.empty(N_MAX)` avec une valeur réelle pour `N_MAX`. Cela produit un message d'erreur de numba, qui sont en général plus difficile à comprendre que les messages d'erreurs de python de base.

Ici on ajoute devant (ligne au-dessus du def ...) la définition des trois fonctions coord, place_libre et remplissage l'indication à numba de compiler ces fonctions :

```
from numba import njit
@njit
def coord(L, R) :
    ...
@njit
def place_libre(n, x, y, x_new, y_new):
    ...
@njit
def remplissage(L, R, MAX_TRIES) :
    ...
```

3.4.2 Liens sur l'empilement compact

En sciences on peut aussi faire des expériences amusantes et les publier dans un journal important comme "Science". Comme cette expérience avec laquelle les chercheurs ont démontrés que l'empilement au hasard des bonbons M&Ms dans une sphère donne un empilement presque aussi dense que l'empilement ordonné le plus compact de sphères de taille égale (empilement cubique à faces centrées ou empilement hexagonal compact) :

`http://www.ncbi.nlm.nih.gov/pubmed/14963324`

(voir aussi sur moodle pour avoir l'article)

Cette bonne performance des bonbons M&Ms est dû à leur forme ellipsoïdale, qui permet un meilleur empilement.

TP 4

Valeurs propres et vecteurs propres : Résolution de l'équation de Schrödinger par un calcul de différences finies

L'équation de Schrödinger $H\psi = E\psi$ est l'équation maîtresse de la mécanique quantique : résoudre un problème de physique quantique revient en général à résoudre cette équation dans le cas considéré. Malheureusement, même dans les cas simples, on se heurte à des problèmes techniques souvent ardus quand on cherche à le faire ; fort souvent même, il n'y a pas de solution analytique du tout ! Dans le présent exercice, on étudiera une méthode simple, peu exigeante sur le plan de la programmation (il s'agit essentiellement de faire appel à un sous-programme de bibliothèque déjà existant) pour tenter de résoudre numériquement des problèmes dont la solution est connue (puits infini, oscillateur harmonique) afin de contrôler la méthode, puis dans des cas où la solution n'est pas connue. Ce sera aussi l'occasion de se familiariser à peu de frais avec cet objet étrange qu'est la fonction d'onde ψ.

On se restreint ici à l'équation de Schrödinger stationnaire (ou non-dépendante du temps) à une particule et à une dimension. Il s'agit donc de chercher les états propres d'une particule dans un potentiel stationnaire [1].

Dans ces conditions, l'équation de Schrödinger s'écrit :

$$-\frac{\hbar^2}{2m}\frac{d^2\psi(x)}{dx^2} + V(x)\,\psi(x) = E\,\psi(x) \tag{4.1}$$

où V et ψ sont des fonctions de l'abscisse x, $V(x)$ est l'énergie potentielle de la particule au point d'abscisse x, $\psi(x)$ la fonction d'onde qui décrit l'état de la particule et E l'énergie associée à cet état ; m est la masse de la particule et $\hbar$ la constante de Planck. Résoudre l'équation (4.1) revient à trouver les fonctions d'ondes $\psi(x)$ et les énergies E pour lesquelles l'équation est vérifiée : il s'agit d'une équation différentielle du second ordre dont on connaît les solutions analytiques dans certains cas ($V = Cst$, $V = \frac{1}{2}kx^2$, ...), mais reste insoluble dans bien des cas d'où la recherche de solutions numériques.

Afin de simplifier les notations, on utilisera un système d'unités *ad hoc* dans lequel $\frac{\hbar^2}{2m} = 1$, donc l'équation (4.1) devient :

$$-\frac{d^2\psi(x)}{dx^2} + V(x)\,\psi(x) = E\,\psi(x) \tag{4.2}$$

1. voir le cours de physique quantique.

4.1 La méthode des différences finies.

Le problème est légèrement différent de celui de la résolution d'une (ou plusieurs) équation(s) du mouvement (voir le problème du patin) où l'on calcule de proche en proche les valeurs que prend la ou les fonctions recherchées : ici, on veut simultanément *toutes* les valeurs de $\psi(x)$ et de surcroît E est aussi une inconnue. Les méthodes de type Runge-Kutta ne conviennent donc pas, il faut en trouver d'autres.

La méthode des différences finies en est une possible. Elle consiste en une discrétisation du problème, c'est-à-dire que l'on fait l'approximation de remplacer la variable continue x par une variable discrète x_i :

$$x_i = -L/2 + i \cdot \delta_x,\ i \in [0, n-1]$$

où i est un indice entier. La droite infinie est évidemment remplacée par un segment fini de longueur $L = (n-1)\delta_x$, centré autour de zéro, on couvre alors l'intervalle $[-L/2, L/2]$.

Remarque : Les réponses aux questions 1 à 7 des deux premières parties sont fournis ici et n'ont pas besoin de figurer dans votre compte-rendu, mais elles doivent être lues et comprises avant de se lancer dans la programmation.

1. Qu'est-ce que cela entraîne, avant même tout calcul, pour n et δ_x ?

Réponse :

δ_x doit être « petit », c'est-à-dire sensiblement plus petit qu'une distance « typique » sur laquelle la fonction d'onde varie sensiblement. Par ailleurs, L doit être « presque » infini, soit, plus grand que le domaine dans lequel la particule peut se déplacer, autrement dit n doit être « grand ».

2. On pose :

$$\begin{aligned} \psi_i &= \psi(x_i) \\ V_i &= V(x_i) \end{aligned}$$

Trouver une expression approchée pour la dérivée première de ψ prise un demi-intervalle au delà de x_i :

$$\frac{d\psi}{dx}(x_i + \frac{\delta_x}{2})$$

ainsi que la même dérivée prise un demi-intervalle en deçà de x_i :

$$\frac{d\psi}{dx}(x_i - \frac{\delta_x}{2})$$

Réponse :

$$\frac{d\psi}{dx}(x_i + \frac{\delta_x}{2}) \sim \frac{\psi_{i+1} - \psi_i}{\delta_x}$$
$$\frac{d\psi}{dx}(x_i - \frac{\delta_x}{2}) \sim \frac{\psi_i - \psi_{i-1}}{\delta_x}$$

3. En déduire une expression approchée pour la dérivée seconde de ψ prise en x_i :

$$\frac{d^2\psi}{dx^2}(x_i)$$

Réponse :

$$\frac{d^2\psi}{dx^2} \sim \frac{\frac{d\psi}{dx}(x_i + \frac{\delta_x}{2}) - \frac{d\psi}{dx}(x_i - \frac{\delta_x}{2})}{\delta_x} = \frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\delta_x^2}$$

4. Écrire l'équation de Schrödinger sous une forme discrétisée.
Réponse :

$$-\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\delta_x^2} + V_i\psi_i = E\psi_i \tag{4.3}$$

4.2 Valeurs propres-vecteurs propres.

On considère maintenant que la fonction d'onde discrétisée est un vecteur :

$$\psi = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_i \\ \vdots \\ \psi_{n-2} \\ \psi_{n-1} \end{pmatrix}$$

5. Écrire l'expression de la matrice $n \times n$ notée **H** telle que l'équation de Schrödinger discrétisée s'écrive maintenant sous forme matricielle :

$$\mathbf{H}\psi = E\psi$$

On appellera **H** l'Hamiltonien discrétisé : ce n'est plus un opérateur comme dans l'équation (4.1), mais une matrice [2].

Cette matrice a-t-elle des propriétés particulières ?
Réponse :
L'équation de Schrödinger discrétisée s'écrit pour tous les ψ_i :

$$\sum_{j=0}^{n-1} H_{ij} \cdot \psi_j = E \cdot \psi_i \tag{4.4}$$

Si on regroupe l'équation 4.3 par les différents ψ_i, on obtient :

$$\begin{aligned} -\frac{\psi_{i+1} - 2\psi_i + \psi_{i-1}}{\delta_x^2} + V_i\psi_i &= E\psi_i \\ -\frac{1}{\delta_x^2}\psi_{i-1} + \left(\frac{2}{\delta_x^2} + V_i\right)\psi_i - \frac{1}{\delta_x^2}\psi_{i+1} &= E\psi_i \end{aligned} \tag{4.5}$$

En comparant l'équation 4.4 avec l'équation 4.5 on obtient :

2. Pour mieux comprendre le lien entre la matrice **H** et l'équation (4.1), il est utile d'écrire l'opérateur hamiltonien en représentation de position : L'opérateur hamiltonien est défini comme $\hat{H} = \frac{\hat{P}^2}{2m} + V(\hat{X})$ avec l'opérateur impulsion $\hat{P}$ et l'opérateur de position $\hat{X}$. On fait alors le choix de la "représentation de position $|x\rangle$" définie par $\hat{X}|x\rangle = x|x\rangle$ où x est maintenant une variable. Dans cette représentation on obtient $\langle x|P|\psi\rangle = -i\hbar\frac{d\psi(x)}{dx}$ et $\langle x|V(X)|\psi\rangle = V(x)\psi(x)$ où $\langle x|\psi\rangle = \psi(x)$ est la fonction d'onde.

$$H_{ii} = \frac{2}{\delta_x^2} + V_i$$
$$H_{i(i-1)} = -\frac{1}{\delta_x^2}$$
$$H_{i(i+1)} = -\frac{1}{\delta_x^2}$$

Donc la matrice **H** a cette forme :

$$\mathbf{H} = \begin{pmatrix} \frac{2}{\delta_x^2} + V_0 & -\frac{1}{\delta_x^2} & 0 & 0 & 0 & \dots & 0 \\ -\frac{1}{\delta_x^2} & \frac{2}{\delta_x^2} + V_1 & -\frac{1}{\delta_x^2} & 0 & 0 & \dots & 0 \\ 0 & -\frac{1}{\delta_x^2} & \frac{2}{\delta_x^2} + V_2 & -\frac{1}{\delta_x^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & 0 & -\frac{1}{\delta_x^2} & \frac{2}{\delta_x^2} + V_i & -\frac{1}{\delta_x^2} & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & -\frac{1}{\delta_x^2} & \frac{2}{\delta_x^2} + V_{n-2} & -\frac{1}{\delta_x^2} \\ 0 & \dots & 0 & 0 & 0 & -\frac{1}{\delta_x^2} & \frac{2}{\delta_x^2} + V_{n-1} \end{pmatrix}$$

C'est une matrice tridiagonale symétrique.

6. Que se passe-t-il en x_0 et x_{n-1} ? On fera l'approximation supplémentaire que $\psi_{-1} = \psi_n = 0$, c'est-à-dire que la fonction d'onde « à l'infini » est nulle. Quel sens physique cela peut-il avoir ?

Réponse :

En $i = 0$ on n'a pas le terme en ψ_{i-1} et en $i = n - 1$, le terme ψ_{i+1} est absent. Cela revient à postuler que la fonction d'onde est nulle aux deux extrémités de l'intervalle fini considéré. La conséquence en est que la particule ne doit pas s'aventurer vers les extrémités de l'intervalle et que donc l'extension de sa fonction d'onde doit rester à peu près centrée et petite devant la taille de l'intervalle. Physiquement, la particule doit donc rester relativement localisée au centre de l'intervalle. Le fait que la fonction d'onde est nulle en dehors de l'intervalle $[-L/2, L/2]$, induit qu'on a toujours un puits carré infini comme potentiel en plus du potentiel qu'on impose avec $V(x)$.

7. Quel type de problème mathématique doit-on maintenant résoudre ? De ce point de vue, que peut on alors dire du vecteur ψ et de E ?

Réponse :

Le problème est maintenant devenu une équation aux valeurs propres : les valeurs que peut prendre E sont les valeurs propres de la matrice **H** et le vecteur ψ est le vecteur propre asssocié à chaque valeur propre. Comme la matrice est $n \times n$, il doit y avoir n valeurs propres et n vecteurs propres solution de l'équation de Schrödinger discrétisée.

4.3 Mise en œuvre numérique.

Il n'est pas question d'écrire une fonction capable de résoudre un problème aux valeurs propres ! Il existe de nombreuses bibliothèques de programmes déjà écrits et testés : ils sont fiables et efficaces, il est inutile de réinventer la roue ! On utilisera ici la bibliothèque scipy.linalg, qui est une collection d'outils pour traiter les problèmes d'algèbre linéaire. La bibliothèque contient différentes fonctions pour diagonaliser des matrices. Pour la diagonalisation des matrices hermitiennes (ou des matrices réelles

symétriques), on peut utiliser la fonction `eigh`. Cependant, ici, nous utiliserons la routine de diagonalisation `eigh_tridiagonal`, qui diagonalise les matrices symétriques tridiagonales :

```
import numpy as np
from scipy.linalg import eigh_tridiagonal
n = 100             # dimension de la matrice
d = np.zeros(n)     # diagonale principale de la matrice
e = np.zeros(n-1)   # diagonale juste au-dessus
...                 # remplissage de d,e
w, v = eigh_tridiagonal(d,e)
```

La fonction `eigh_tridiagonal(d,e)` prend la diagonale `d` et les éléments au-dessus de la diagonale `e` d'une matrice tridiagonale symétrique comme argument, puis elle diagonalise la matrice et renvoie les valeurs propres dans le vecteur `w` et les vecteurs propres dans la matrice `v`. Plus précisément, le vecteur `w` contient les valeurs propres, par ordre croissant, chacune étant répétée en fonction de sa multiplicité.

Il ne reste plus qu'à remplir la diagonale `d` et la sous-diagonale `e` qui représentent $\mathbf{H}$ en forme discrète. La taille du Hamiltonien `n*n` dépend du nombre d'éléments du vecteur ψ. A noter qu'au lieu d'initialiser et de remplir ensuite les vecteurs `d` et `e`, on peut directement définir ces vecteurs correctement dès le départ.

On remarquera que les fonctions d'ondes calculées ne sont pas normalisées comme les fonctions d'ondes théoriques avec

$$\int \psi^2 dx = 1$$

On peut le faire, si souhaité, avec cette fonction par exemple sur la matrice `v` des vecteurs propres :

```
def normalize(m,delta_x):
    m /= np.linalg.norm(m, axis=0)      # normaliser les colonnes
    m /= np.sqrt(delta_x)               # diviser par 1/sqrt(dx)
    return m
```

Les colonnes de `v` sont déjà normalisées, mais pour être sûrs nous le faisons encore dans la fonction `nomalize`. La multiplication avec le scalaire $1/\sqrt{\delta x}$ peut être effectuée pour tous les éléments de la matrice avec une seule commande.

Lors de ce TP seront générés beaucoup de fichiers *.pdf différents en fonction du potentiel et des paramètres `L` et `n`. Pour garder une trace de quel fichier correspond à quoi, il est utile de directement inclure les valeurs des paramètres dans le nom de fichier. Pour faire cela d'une manière automatique, on peut utiliser `%i` dans le nom de fichier, qui sera remplacé par `L` ou `n` pour définir le nom de fichier. Voir cet exemple :

```
import matplotlib.pyplot as plt
plt.plot(...)     # tracer les résultats
plt.savefig('q4_harm_valeurs_n%i_L%i.pdf'%(n,L))
```

Ici le début du nom de fichier (`q4_harm_valeurs`) indique qu'il s'agit de valeurs propres de la question 4 pour un potentiel harmonique, puis on spécifie les paramètres `n` et `L`. Ce qui donne par exemple : `q4_harm_valeurs_n100_L20.pdf`. On utilise ici la même syntaxe que pour les `print()`, mais cette fois-ci, nous avons fourni deux arguments (`%i`), c'est pourquoi il faut fournir un tuple de variables entières juste après (`%(n,L)`).

8. Écrire une fonction qui rend la valeur $V(x)$. Pour le puits infini, il suffit de mettre 0, pour le potentiel harmonique, x^2. Lorsqu'on voudra ensuite changer la forme du potentiel, il suffira de modifier cette fonction, à l'exclusion du reste du programme. Cette fonction est alors de cette forme :

```
def potentiel(x):
    return ...
```

9. Écrire un programme principal qui :

1. définisse un intervalle $[-L/2, L/2]$, le divise en n segments de longueur δx et remplisse les tableaux `x` (contenant les x_i) et `V` (contenant les valeurs de potentiel V_i). On peut soit utiliser des tableaux numpy classiques (structure de données `ndarray`). Prenez ces valeurs pour commencer : $L = 5$ et $n = 100$.
2. crée les tableaux 1D `d,e` et les remplisse avec les éléments diagonaux et sous-diagonaux de $\mathbf{H}$.
3. en recherche les valeurs propres et les vecteurs propres.

4.4 Étude numérique.

4.4.1 Puits carré infini.

Il est facile de définir un puits carré infini : il suffit de fixer la valeur du potentiel à zéro partout sur l'intervalle ; comme la fonction d'onde est toujours nulle en dehors de l'intervalle, cela revient à ce que le potentiel soit infini en dehors dudit intervalle.

Rappels théoriques[3] *:*

Si on considère une particule dans un puits carré infini de largeur L, les fonctions d'onde solutions de l'équation de Schrödinger indépendante du temps sont nulles en dehors du puits et ont la forme suivante dans le puits

$$\psi_p(x) = \sqrt{\frac{2}{L}} \sin \frac{(p+1)\pi(x+\frac{L}{2})}{L}$$

Les énergies de ces états s'écrivent :

$$E_p = \frac{\hbar^2}{2m}\left(\frac{\pi(p+1)}{L}\right)^2$$

Rappel : on choisit ici $\frac{\hbar^2}{2m} = 1$.

10. Essayer le programme avec $n = 100$ et $L = 5$, afin de comparer le résultat obtenu avec le résultat théorique.

Tracer la courbe E_p en fonction de p avec, sur le même graphe, la courbe théorique. Vérifier l'accord avec la courbe théorique.

Tracer également les premières fonctions d'onde ($p \in [0, 2]$) obtenues par votre programme. Pour cela tracer sur le même graphe les premières ($p \in [0, 2]$) fonctions d'onde au carré, décalées verticalement d'une valeur proportionnelle à leur énergie E_p.

Sur deux autres graphes comparer l'accord entre les fonctions d'onde numériques et théoriques pour $p = 1$ et un ordre plus élevé dans le domaine où l'accord pour E_p n'est pas bon, par exemple $p = 55$ (un graphe par valeur de p). Faites votre diagnostic.

3. Reportez-vous au cours de physique quantique.

4.4.2 Potentiel harmonique.

Rappels théoriques.

Le deuxième cas dont la solution est connue est le potentiel harmonique [4] :

$$V(x) = \frac{1}{2}kx^2 = \frac{m}{2}\omega^2 x^2, \quad \omega = \sqrt{\frac{k}{m}}$$

Les énergies propres s'écrivent maintenant :

$$E_p = \left(p + \frac{1}{2}\right)\hbar\omega, \quad p = 0, 1, 2, 3, \ldots$$

L'état fondamental, pour $p = 0$ a donc une énergie $E_0 = \hbar\omega/2$.

Les fonctions d'onde deviennent :

$$\psi_p(x) = \frac{1}{\sqrt{\sigma_0\sqrt{\pi}2^p p!}} H_p(\frac{x}{\sigma_0})\mathrm{e}^{-\frac{x^2}{2\sigma_0^2}}$$

où $\sigma_0 = \sqrt{\dfrac{\hbar}{m\omega}}$ et H_p est un polynôme de Hermite.

11. Modifier votre programme pour étudier le potentiel harmonique : on prendra $k = 1$, soit $\hbar\omega = \sqrt{2m}\sqrt{k/m} = \sqrt{2}$.

Répéter les opérations précédentes avec $n = 100$ et $L = 5$, tracer les énergies avec la courbe théorique. Vue la complexité de la formule des fonctions d'ondes théoriques, on ne les tracera pas ici. Tracez les trois premières fonctions d'onde numériques comme en haut en décalant chaque fonction d'une valeur proportionnel à son énergie. Ajouter à ce graphe aussi le potentiel $V(x)$. N'y a-t-il pas une complication supplémentaire par rapport au cas précédent ?

Refaire le même calcul, toujours avec $n = 100$ mais $L = 20$. Diagnostic.

4.4.3 Double puits.

L'intérêt de cet exercice n'est bien sûr pas de chercher à résoudre un problème - le puits infini ou l'oscillateur harmonique - dont la solution analytique est connue : cela permet juste de tester le programme que l'on a écrit et d'illustrer le cours de physique quantique.

Toutefois, le même programme, à très peu de choses près, permet de résoudre des problèmes sensiblement plus compliqués, pour lesquels les solutions analytiques sont pour l'essentiel inaccessibles. Remplacer le potentiel harmonique par :

$$V(x) = a(x - r_1)(x - r_2)(x - r_3)(x - r_4)$$

soit un polynôme de degré 4, dont les racines sont r_1, r_2, r_3, r_4.

Prenez $n = 1000$ et $L = 20$.

12. Comme précédemment, tracer sur un seul graphe les solutions obtenues des premiers états pour $a = 1$ et $r_1 = r_2 = r_3 = r_4 = 0$, ainsi que le potentiel $V(x)$. Commenter.

13. Tracer les solutions obtenues pour $a = 1$ et $r_1 = -2$, $r_2 = -0.5$, $r_3 = 0.5$, $r_4 = 2$ (double puits symétrique). Commenter.

4. Se reporter de nouveau au cours de physique quantique.

14. Tracer les solutions obtenues pour $a = 400$ et $r_1 = -2$, $r_2 = -0.5$, $r_3 = 0.5$, $r_4 = 2$ (double puits symétrique plus profond). Commenter et, bien sûr, comparer avec le cas précédent.

15. Tracer les solutions obtenues pour $a = 1$ et $r_1 = -2$, $r_2 = -0.5$, $r_3 = 0$, $r_4 = 2$ (double puits dissymétrique). Commenter.

4.5 Pour aller plus loin : L'électron.

Jusqu'à présent la particule étudiée n'était pas identifiée, ça pouvait être n'importe quel type de particule.

On peut imaginer maintenant qu'il s'agit d'un électron soumis au potentiel dû aux atomes environnants, noyaux et électrons associés, afin de déterminer les propriétés électroniques du milieu considéré. A priori, ce potentiel devrait s'écrire

$$V(x) = \frac{e^2}{4\pi\epsilon_0}\left(-\sum_i \frac{Z_i}{|x - X_i|} + \sum_j \frac{1}{|x - x_j|}\right)$$

où Z_i et X_i sont les numéro atomique et position de l'atome i et x_j sont les positions de électrons qui accompagnent les noyaux. Malheureusement, en dépit de la simplicité apparente de cette expression, ce problème est d'une difficulté redoutable ! En effet, on ne peut pas découpler le mouvement d'un électron de celui de tous les autres et il faudrait donc écrire une fonction d'onde globale de tous les électrons [5] : $\psi(x_1, x_2, x_3, \ldots, x_{n_e})$.

On peut toutefois souvent faire l'approximation où l'on considère *un* électron dans un champ *effectif* créé par les noyaux et leurs électrons. On perd ainsi les effets de l'interaction explicite électron-électron, mais si les autres électrons sont « bien accrochés » à leur noyau, ce n'est pas toujours très grave. . .

On peut, par exemple, tenter :

$$V(x) = \sum_i -V_0\, \mathrm{e}^{-\frac{|x-X_i|}{\rho}}$$

On pourra prendre 10 atomes, $n = 1000$, $L = 20$, $V_0 = 400$ et $\rho = 0.5$. Les atomes seront également répartis entre $x = -6$ et $x = 6$.

Tracer les 20 premières énergies propres et 11 ou 12 densités de probabilité (sur le même graphe que l'énergie potentielle).

Commenter.

Tracer les densités de probabilité pour $V_0 = 40$ et commenter.

4.6 Conclusion.

On a, sans coup férir, diagonalisé des matrices (1000×1000) en comptant sur la capacité de calcul de l'ordinateur : c'est ce qu'on appelle un calcul par « force brute ». Compte tenu du temps que cela prend, que pensez-vous d'un problème tridimensionnel à un électron (et donc une matrice ($1000^3 \times 1000^3$) à traiter), et a fortiori d'un problème tridimensionnel à quelques centaines d'électron (en gros, une molécule simple. . .) ?

5. l'approximation dite de « Born-Oppenheimer » permet, en général, de considérer les noyaux comme immobiles. . .