**UNIVERSITÉ PARIS-SACLAY 2021-2022 première session d'examen**

**Licence 3 et Magistère 1ère année de Physique Fondamentale**

# Examen d'informatique

**jeudi 28 avril 2022 13h45 à 15h45 (2 heures)**

— *Aucun document n'est autorisé.*

— *Les programmes doivent être écrits en C/C++.*

— *Respecter les notations de l'énoncé : par exemple la variable notée $n_t$ dans l'énoncé devra être écrite* `nt` *dans un programme, $n''$ devra être écrite* `ns`.

— *L'utilisation de fonctions est laissée à l'appréciation de l'étudiant, sauf dans les cas où elle est imposée.*

— *On suppose que tous les* `#include<iostream> ... #include<math.h> using namespace std;` *nécessaires sont sous-entendus, il n'y a pas à les écrire.*

— *Ne pas faire lire les données au clavier ou dans un fichier par un* `cin >>`*, ou l'équivalent pour un fichier, sauf si cela est explicitement demandé. Par défaut, les données seront donc fournies dans le programme lui-même, par des instructions du type :*

```
dx=0.01; a=1.7; b=1.1;
```

*Si la valeur d'une variable n'est pas explicitement donnée, vous pouvez écrire* `dx=...;`

— *Tous les tableaux dont il s'agit dans l'énoncé sont des tableaux dynamiques, à déclarer avec un ou plusieurs* `malloc`.

— *Dans chaque exercice on suppose que le programme principal (s'il y en a un) et les fonctions (s'il y en a) sont écrits dans un unique fichier.*

— *L'examen consiste en 3 exercices, qui sont indépendants les uns des autres.*

— ***Le nombre de points (sur un total de 21) attribué à chaque question est indiqué entre parenthèses. Un point est réservé à la qualité de la présentation, pour un total de 22 points.***

— *Le corrigé pourra être consulté sur le site du cours ultérieurement.*

## 1. Réfraction de rayons lumineux

Lorsqu'un rayon lumineux atteint l'interface entre deux milieux optiques différents, sa direction de propagation est changée. La modification de sa direction de propagation est contrôlée par les indices de chaque milieu. L'indice optique d'un milieu transparent dépend de la longueur d'onde du rayon qui le traverse. Cette dépendance est mesurable empiriquement, et peut souvent s'écrire par la formule :

$$n(\lambda) = A_0 + \frac{A_1}{\lambda^2} + \frac{A_2}{\lambda^4} + \cdots = \sum_{i=0}^{N} \frac{A_i}{\lambda^{2i}} \tag{1}$$

**Question 1 : (1 point)**

Écrire une fonction de prototype `double indice(double* coeff, int nb_coeff, double lambda)` qui calcule, à la longueur d'onde `lambda`, l'indice d'un milieu dont les `nb_coeff` premiers coefficients $A_i$ sont stockés dans le tableau `coeff`. On rappelle la fonction du C `pow(a,b)` qui calcule $a^b$.

*Réponse à la question 1 :*

```
double indice(double* coeff, int nb_coeff, double lambda) {
  int i;
  double ind = 0.;
  for(i = 0; i < nb_coeff; i++)
    ind += coeff[i] * pow(lambda, -2*i);
  return ind;
}
```

La réfraction d'un angle passant d'un milieu d'indice $n_1$ à un milieu d'indice $n_2$, arrivant avec un angle d'incidence $\theta_i$ sur l'interface, est donnée par la loi bien connue de Snell-Descartes :

$$n_1 \sin(\theta_i) = n_2 \sin(\theta_r) \tag{2}$$

où $\theta_r$ désigne l'angle du rayon réfracté (tous les angles sont exprimés en radians).

**Question 2 : (0.75 point)**

On se limite dans cette question au cas où le rayon arrive de l'air, qu'on modélise par un milieu d'indice $n_1 = 1$, vers un milieu d'indice $n$. Écrire une fonction nommée `angle_depuis_air` qui prend en argument l'indice du nouveau milieu, un tableau d'angles d'incidences et la longueur de ce tableau, et affiche à l'écran pour chacun l'angle du rayon réfracté. On rappelle que l'inverse du sinus est calculé en C avec la fonction `asin`.

*Réponse à la question 2 :*

```
void angle_depuis_air(double ind, double* angles, int nb_angle) {
  int i;
  for(i = 0; i < nb_angle; i++)
    cout << angles[i] << " -> " << asin(sin(angles[i]) / ind) << endl;
}
```

Dans le cas où un rayon lumineux arrive d'un milieu d'indice plus élevé à un milieu d'indice plus faible ($n_1 > n_2$), la loi de Snell-Descartes peut ne pas avoir de solution si l'angle d'incidence est trop grand. C'est ce qu'on appelle le cas de réflexion totale interne, où il n'y a pas de rayon réfracté, et qui arrive au-delà de l'angle critique $\theta_c$ donné par :

$$\frac{n_1}{n_2} \sin(\theta_c) = 1 \qquad \Leftrightarrow \qquad \theta_c = \arcsin\left(\frac{n_2}{n_1}\right) \tag{3}$$

**Question 3 : (0.25 point)**

Écrire une fonction `ref_totale` qui prend en entrée l'indice du milieu d'incidence et renvoit l'angle critique pour un rayon passant de ce milieu à un milieu d'indice $n = 1$.

*Réponse à la question 3 :*

```
double ref_totale(double ind) {
  return asin(1./ind);
}
```

**Question 4 : (1 point)**

Écrire une fonction `angle_vers_air` qui prend en argument l'indice du milieu d'incidence, un tableau d'angles d'incidences et la longueur de ce tableau, et affiche à l'écran pour chacun l'angle du rayon réfracté s'il y en a un, ou `"réflexion totale"` sinon. On suppose que le rayon passe du milieu d'incidence à un milieu d'indice $n = 1$.

*Réponse à la question 4 :*

```
void angle_vers_air(double ind, double* angles, int nb_angle) {
  int i;
  double tc = ref_totale(ind);
  for(i = 0; i < nb_angle; i++)
    if(angles[i] < tc)
      cout << angles[i] << " -> " << asin(sin(angles[i]) * ind) << endl;
    else
      cout << angles[i] << " -> reflexion totale" << endl;
}
```

**Question 5 : (2 points)**

Écrire un programme principal (`int main()`) qui lit les valeurs des coefficients de l'indice dans un fichier *coefficients.dat*, dont la première ligne est le nombre de coefficients stockés `nb_coeff`, et les stocke dans un tableau à un indice `coeff` (à allouer avec `malloc`). Puis il calculera l'indice `ind` du milieu caractérisé par ces coefficients pour une longueur d'onde $\lambda = 5 \cdot 10^{-7}$ m.

Le programme devra ensuite demander (avec la commande `cin`) un angle minimal, un angle maximal, et un nombre d'angles `nb_angle`. Le programme créera un autre tableau pour stocker tous les angles d'incidences, calculera ces `nb_angle` angles répartis linéairement entre les angles min et max (bords inclus), puis les mettra dans le tableau.

Le programme affichera alors les angles des rayons réfractés (ou `"réflexion totale"` s'il y a lieu) dans les deux cas précédents (interface depuis ou vers un milieu d'indice $n = 1$, l'indice de l'autre milieu étant `ind`) pour tous les angles stockés dans le tableau. Vous ferez bien sûr appel aux fonctions écrites dans les questions précédentes.

*Réponse à la question 5 :*

```
int main() {
  int i, nb_coeff, nb_angle;
  double lambda = 5e-7;
  double ind, ang_max, ang_min;
  double *coeff, *angles;
  fstream fich;

  fich.open("coefficients.dat", ios::in);
  fich >> nb_coeff;
  coeff = (double*)malloc(nb_coeff * sizeof(double));
  for(i = 0; i < nb_coeff; i++)
    fich >> coeff[i];
  fich.close();
  ind = indice(coeff, nb_coeff, lambda);
  free(coeff);

  cout << "Angle minimal, angle maximal, et nombre d'angles ? ";
  cin >> ang_min >> ang_max >> nb_angle;
  angles = (double*)malloc(nb_angle * sizeof(double));
  for(i = 0; i < nb_angle; i++)
    angles[i] = ang_min + (ang_max - ang_min)/(nb_angle-1) * i;

  cout << "Refraction depuis l'air" << endl;
  angle_depuis_air(ind, angles, nb_angle);
  cout << "Refraction vers l'air" << endl;
  angle_vers_air(ind, angles, nb_angle);
  free(angles);
  return 0;
}
```

## 2. Pointeurs – Reversi

Le but de cet exercice est d'implémenter quelques fonctions qui seraient utiles à la réalisation d'un jeu Othello ou Reversi. [1] Ce jeu se joue sur un tablier de 64 cases ($8 \times 8$), avec des pions bicolores, une face noire et une face blanche. L'objectif à chaque coup est de retourner une partie des pions adverses. On représentera l'état du tablier avec un tableau d'entiers à deux dimensions, 0 représentant une case libre, -1 lorsqu'elle est occupée par un pion noir et $+1$ pour un pion blanc. Dans tout l'exercice, on utilisera une variable globale de type entier `int N = 8;` qui représente le nombre de lignes et de colonnes du tablier.

**Question 1 : (0.75 point)**

Écrire le corps d'une fonction C sans argument `int** reversi_alloc()` qui renvoie le tableau qui pourra contenir l'état du tablier. On utilisera la fonction `malloc(...)` pour allouer la mémoire.

*Réponse à la question 1 :*

```
int** reversi_alloc() {
  int i;
  int** T = (int**)malloc(N * sizeof(int*));
  for(i = 0; i < N; i++)
    T[i] = (int*)malloc(N * sizeof(int));
  return T;
}
```

**Question 2 : (0.75 point)**

Le jeu commence avec 4 pions disposés sur le tablier selon le schéma ci-dessous. Définir le prototype et écrire le corps d'une fonction `void reversi_init(...)` qui accepte comme argument le tableau représentant le tablier et l'initialise pour le début du jeu.

```
   a  b  c  d  e  f  g  h
1
2
3
4           B  N
5           N  B
6
7
8
```

*Réponse à la question 2 :*

```
void reversi_init(int** T) {
  int i, j;
  for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
      T[i][j] = 0;
  T[3][3] = 1; T[3][4] = -1;
  T[4][3] = -1; T[4][4] = 1;
}
```

**Question 3 : (1.5 points)**

Soit `T` le pointeur de type `int**` qui pointe sur le tableau rempli à la question précédente. Si l'on faisait afficher les expressions suivantes à l'écran, indiquer ce qui serait affiché : une valeur (indiquer laquelle), une adresse ou une erreur.

1. `T[3][3]`
2. `&(T[3][3])`
3. `*(T[3][3])`
4. `*&(T[3][3])`
5. `(*&T)[3][3]`
6. `(*T)[3]`
7. `**(T+3)`
8. `*(&(T[3])+3)`

1. Vous pourrez consulter la page wikipedia `https://fr.wikipedia.org/wiki/Othello_(jeu)` pour connaître les règles du jeu.

*Réponse à la question 3 :*

1. 1
2. adresse
3. erreur
4. 1 (= `T[3][3]`)
5. 1 (= `T[3][3]`)
6. 0 (= `T[0][3]`)
7. 0 (= `T[3][0]`)
8. adresse (= `T[6]`)

**Question 4 : (1 point)**

Définir le prototype et écrire le corps d'une fonction `... reversi_count(...)` qui renvoie comme valeur de retour le nombre de cases vides, ainsi que les nombres de cases occupées par des pions blancs et des pions noirs dans deux variables fournies comme arguments de la fonction.

*Réponse à la question 4 :*

```
int reversi_count(int** T, int* blanc, int* noir) {
  int i, j, vide = 0;
  *blanc = 0; *noir = 0;
  for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
      if (T[i][j] == 1)
        (*blanc)++;
      else if (T[i][j] == -1)
        (*noir)++;
      else
        vide++;
  return vide;
}
```

**Question 5 : (1 point)**

Écrire le programme principal `int main()` qui déclare les variables nécessaires, crée et initialise le tablier, calcule le nombre de cases vides et occupées et affiche ces nombres de cases à l'écran.

*Réponse à la question 5 :*

```
int main() {
  int i, vide, blanc, noir;
  int** T = reversi_alloc();
  reversi_init(T);
  vide = reversi_count(T, &blanc, &noir);
  cout << "Comptage de cases : Vides="<<vide<<" Blanches="<<blanc<<" Noires="<<noir<<endl;
  for(i = 0; i < N; i++)
    free(T[i]);
  free(T);
  return 0;
}
```

## 3. Potentiel de Lennard-Jones

On considère $N$ particules identiques dans un plan dont les coordonnées sont $\vec{r}_i = (x_i, y_i)$ avec $i = 0, 1, \ldots, N-1$. Ces particules interagissent par le potentiel de Lennard-Jones, ce qui veut dire que la force (divisée par la masse) de la particule $j$ sur la particule $i$ est

$$\frac{\vec{F}_{i,j}}{m} = 24E\left(2\frac{d^{12}}{|\vec{r}_i - \vec{r}_j|^{14}} - \frac{d^6}{|\vec{r}_i - \vec{r}_j|^8}\right)(\vec{r}_i - \vec{r}_j) \tag{4}$$

où $d$ et $E$ sont deux constantes positives. Chaque particule ressent l'interaction avec les $N-1$ autres particules.

On va faire évoluer ce système de $N$ particules à partir d'une configuration initiale en utilisant la méthode d'Euler. Pour pouvoir utiliser la méthode d'Euler vous utiliserez un tableau à une dimension nommé `q` de taille $4N$ qui représente un vecteur $\vec{q}$ organisé comme suivant :

$$\vec{q} = (x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}, \dot{x}_0, \dot{y}_0, \dot{x}_1, \dot{y}_1, \ldots, \dot{x}_{N-1}, \dot{y}_{N-1}) \tag{5}$$

La configuration initiale est définie comme suit. On tire au hasard les positions initiales telles qu'elles soient distribuées uniformément dans le cercle de rayon $R$ dont le centre est l'origine spatiale. On tire au hasard les vitesses initiales selon une distribution gaussienne :

$$f(v) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(\frac{-v^2}{2\sigma^2}\right) \tag{6}$$

(séparément pour chaque $\dot{x}_i$ et chaque $\dot{y}_i$). La constante positive $\sigma$ est l'écart type de la distribution.

**Question 1 : (1.5 points)**

**a)** Déclarer les cinq constantes $N$, $d$, $E$, $R$ et $\sigma$ comme variables globales.

**b)** Écrire une fonction avec le prototype : `double* inipos()` qui déclare et alloue le tableau `q`, puis remplit les positions initiales des particules selon les instructions ci-dessus, et enfin renvoie ce tableau.

Pour rappel, pour avoir des valeurs aléatoires distribuées uniformément dans un cercle avec rayon $R$ centré en $O$, on tire des coordonnées polaires $(r, \theta)$ avec $r = R\sqrt{z}$, où $z$ est de densité uniforme sur $[0, 1]$, et $\theta$ uniforme sur $[0, 2\pi]$.

*Réponse à la question 1 :*

```
int N = ...;
double d = ..., E = ..., R = ..., sigma = ...;

double* inipos() {
  double r, th;
  int i;
  double* q = (double*)malloc(4*N * sizeof(double));
  for(i = 0; i < 2*N; i += 2) {
    r = R * sqrt(drand48());
    th = 2 * M_PI * drand48();
    q[i] = r * cos(th);
    q[i+1] = r * sin(th);
  }
  return q;
}
```

**Question 2 : (1 point)**

Écrire une fonction avec le prototype : `void inivit(double* q)` qui remplit les vitesses initiales des particules dans le tableau `q` selon les instructions ci-dessus. Vous utiliserez la méthode de rejet de Von Neumann en supposant que la fonction gaussienne est nulle en dehors de l'intervalle $[-5\sigma, 5\sigma]$.

On rappelle que la méthode de rejet pour générer des valeurs aléatoires $v_x$ (ou $v_y$) de densité de probabilité $f(v)$ dans un intervalle $[a, b]$ marche comme suivant : on tire une valeur aléatoire $v_x$ de densité uniforme sur $[a, b]$ et une valeur $z$ de densité uniforme sur $[0, M]$ (avec $M$ la valeur supérieure de $f(v)$ sur $[a, b]$), et on accepte $v_x$ seulement si le point $(v_x, z)$ est au-dessous de la courbe $f(v)$, c'est-à-dire si $z \leq f(v_x)$.

*Réponse à la question 2 :*

```
void inivit(double* q) {
  double v, z;
  int i;
  for(i = 2*N; i < 4*N; i++) {
```

```
    do {
      v = 10.*sigma * drand48() - 5.*sigma;
      z = drand48();
    } while(z > exp(-v*v/(2.*sigma*sigma)));
    q[i] = v;
  }
}
```

**Question 3 : (1 point)**

Écrire une fonction avec le prototype :

```
void euler(void (*syst)(double*,double*), double* q, double dt)
```

qui fait une itération de la méthode d'Euler. Vous pourrez supposer que `syst` pointe sur une fonction dont les deux arguments sont le tableau `q` et sa dérivée par rapport au temps `qp` et qui spécifie le système des équations différentielles en calculant `qp` à partir de `q` (vous allez écrire cette fonction dans les questions suivantes).

*Réponse à la question 3 :*

```
void euler(void (*syst)(double*,double*), double* q, double dt) {
  int i;
  double* qp = (double*)malloc(4*N * sizeof(double));
  syst(q, qp);
  for (i = 0; i < 4*N; i++)
    q[i] = q[i] + dt * qp[i];
  free(qp);
}
```

**Question 4 : (1.5 points)**

Dans cette question vous allez écrire le début de la fonction avec le prototype :

```
void systeme(double* q, double* qp)
```

qui va contenir le système des équations différentielles. Pour commencer, écrire les instructions pour déclarer, allouer, remplir et (à la fin de la fonction) libérer un tableau nommé `r2` à deux dimensions défini tel que l'élément $(i, j)$ du tableau contienne la distance au carré entre les deux particules $i$ et $j$.

*Réponse à la question 4 :*

```
void systeme(double* q, double* qp) {
  int i, j;
  double** r2 = (double**)malloc(N * sizeof(double*));
  for(i = 0; i < N; i++) {
    r2[i] = (double*)malloc(N * sizeof(double));
    for(j = 0; j < N; j++)
      if(j > i)
        r2[i][j] = pow(q[2*i]-q[2*j], 2) + pow(q[2*i+1]-q[2*j+1], 2);
      else if(j < i)
        r2[i][j] = r2[j][i];
  } // On n'a pas besoin de remplir les elements i==j, qui ne seront pas utilises

  // Ici les instructions de la question suivante

  for(i = 0; i < N; i++)
    free(r2[i]);
  free(r2);
}
```

**Question 5 : (2 points)**

Terminer la fonction de la question précédente en écrivant les instructions qui calculent les éléments de `qp` à partir des éléments de `q` en utilisant le principe fondamental de la dynamique avec les forces spécifiées en (4).

*Réponse à la question 5 :*

```
  int i0;
  double temp;
  for(i = 0; i < 2*N; i++)
    qp[i] = q[i+2*N];
```

```
  for(i = 2*N; i < 4*N; i += 2) {
    qp[i] = 0.; qp[i+1] = 0.;
    i0 = (i-2*N)/2;
    for(j = 0; j < N; j++)
      if(j != i0) {
        temp = 24*E*(2*pow(d,12)/pow(r2[i0][j],7)-pow(d,6)/pow(r2[i0][j],4));
        qp[i] += temp * (q[2*i0]-q[2*j]);       // ligne a changer en question 7
        qp[i+1] += temp * (q[2*i0+1]-q[2*j+1]); // ligne a changer en question 7
      }
  }
```

**Question 6 : (1.5 points)**

Écrire la fonction `main` telle que le programme soit complet. Elle doit faire le calcul demandé en faisant appel aux fonctions des questions précédentes. On suivra le système pendant $N_t$ étapes entre $t = 0$ et $t = t_{\text{fin}}$ (bords inclus). Les positions de toutes les particules à chaque instant doivent être écrites dans un fichier nommé *LJ.res*, avec sur chaque ligne : $t\ x_0\ y_0\ x_1\ y_1\ \ldots\ x_{N-1}\ y_{N-1}$.

*Réponse à la question 6 :*

```
int main() {
  double* q;
  int i, j, Nt = ...;
  double t = 0, tfin = ..., dt = (tfin-t)/(Nt-1);
  fstream fich;

  srand48(time(NULL));
  q = inipos();
  inivit(q);
  fich.open("LJ.res", ios::out);
  for(i = 0; i < Nt; i++) {
    fich << t << " ";
    for(j = 0; j < 2*N; j++)
      fich << q[j] << " ";
    fich << endl;

    // Ici les instructions de la question suivante

    euler(systeme,q,dt);
    t += dt;
  }
  fich.close();
  free(q);
  return 0;
}
```

**Question 7 : (1.5 points)**

Pour que le système puisse évoluer vers un état d'équilibre, on ajoute une force de frottement pour chaque particule $i$ :

$$\frac{\vec{F}_i^{\text{frot}}}{m} = -f\dot{\vec{r}}_i \tag{7}$$

avec $f$ une constante positive, qui est déclarée comme variable globale.

**a)** Ajouter les instructions nécessaires au programme : la déclaration de $f$ et les modifications de la fonction `systeme` de la question 5 pour prendre en compte les forces de frottement.

**b)** Définir le prototype et écrire le corps d'une fonction nommée `test` qui renvoie un entier qui vaut 1 si toutes les vitesses $|\dot{\vec{r}}_i| < \epsilon$ et 0 sinon, avec $\epsilon$ une constante positive.

**c)** Ajouter les instructions nécessaires au programme principal `main` afin que le calcul s'arrête si l'équilibre est atteint avant $t_{\text{fin}}$ (l'équilibre est défini par la condition sur les vitesses donnée ci-dessus). Il doit afficher un message à l'écran dans ce cas avec l'indication du temps correspondant. À noter que le calcul doit aussi toujours s'arrêter à $t_{\text{fin}}$ après $N_t$ étapes.

*Réponse à la question 7 :*

```
double f = ..., eps = ...;
// Il n'etait pas demande explicitement de declarer eps comme une variable globale,
// d'autres solutions sont possibles

int test(double* q) {
  int i, k;
  k = 1;
  for(i = 2*N; i < 4*N; i += 2)
    if(pow(q[i],2) + pow(q[i+1],2) >= eps*eps) {
      k = 0;
      break;
    }
  return k;
}
```

On change les deux lignes, qui étaient indiquées dans la question 5, comme suivant :

```
        qp[i] += temp * (q[2*i0]-q[2*j]) - f*q[i];
        qp[i+1] += temp * (q[2*i0+1]-q[2*j+1]) - f*q[i+1];
```

On ajoute à l'endroit indiqué de la fonction `main` :

```
    if(test(q) == 1) {
      cout << "Equilibre atteint pour t = " << t << endl;
      break;
    }
```

**Question 8 : (1 point)**

En fait la distance initiale entre les particules ne doit pas être inférieure à $d$. Modifier la fonction `inipos` de la question 1 pour prendre en compte cette condition. Il faut donc tester pour chaque nouvelle particule la distance par rapport à toutes les particules précédentes et tirer une nouvelle particule si elle ne convient pas.

*Réponse à la question 8 :*

La nouvelle version de la fonction `inipos` est :

```
double* inipos() {
  double r, th;
  int i, j, k;
  double* q = (double*)malloc(4*N * sizeof(double));
  for(i = 0; i < 2*N; i += 2)
    do {
      r = R * sqrt(drand48());
      th = 2 * M_PI * drand48();
      q[i] = r * cos(th);
      q[i+1] = r * sin(th);
      k = 0;
      for(j = 0; j < i; j += 2)
        if(pow(q[i]-q[j],2) + pow(q[i+1]-q[j+1],2) <= d*d) {
          k = 1;
          break;
        }
    } while(k == 1);
  return q;
}
```