UNIVERSITÉ PARIS-SACLAY 2024-2025 première session d'examen

Licence 3 et Magistère 1ère année de Physique Fondamentale

Examen d'informatique

lundi 5 mai 2025 13h45 à 16h15 (durée : 2h30)

— *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;
```

— *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.*

— *Les 4 exercices sont indépendants les uns des autres.*

— *Le nombre de points (sur un total de 20.5) attribué à chaque question est indiqué entre parenthèses (c'est un barème indicatif). Un point est réservé à la qualité de la présentation, pour un total de 21.5 points.*

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

1. Etude de la déflexion d'un faisceau de rubidium

On considère un déflecteur d'angle de rotation $\theta = 45°$ et de rayon de courbure $\rho = 0,2$ m, traversé par un faisceau de particules de rubidium-85 (c'est-à-dire avec une masse de 85 unités de masse atomique) monochargé (c'est-à-dire avec une charge de $1,6 \times 10^{-19}$ C) et d'énergie cinétique $E_{\text{cin}} = 30$ keV. On rappelle les formules relativistes suivantes :

$$E_{\text{tot}} = \gamma mc^2 = E_{\text{cin}} + E_{\text{mass}} \qquad \text{avec} \qquad E_{\text{mass}} = mc^2 \qquad \text{et} \qquad \gamma = 1/\sqrt{1-\beta^2} \tag{1}$$

On rappelle également qu'une unité de masse atomique vaut $1u \approx 931,5$ MeV/c^2 avec $c = 3 \times 10^8$ m/s.

Question 1 : (0.75 point)

Écrire une fonction C, ayant comme prototype : `double beta_relat(double Ecin, double Emass)`, qui renvoie le facteur relativiste $\beta = v/c$.

Réponse à la question 1 :

```
double beta_relat(double Ecin, double Emass) {
// On suppose que Ecin et Emass soient specifiees en meme unite
  double Etot, gamma, beta;
  Etot = Ecin + Emass;
  gamma = Etot / Emass;
  beta = sqrt(1-1/(gamma*gamma));
  return beta;
}
```

Question 2 : (0.5 point)

Écrire une fonction avec pour prototype : `double rigidite_mag(double Ecin, double Emass, double beta)` qui renvoie la rigidité magnétique $B\rho$ (produit du champ magnétique et du rayon de courbure) du faisceau, définie par $B\rho = p/q$ avec $p = \gamma mv$ l'impulsion et q la charge électrique. Pour un faisceau monochargé cette expression devient $B\rho = \beta E_{\text{tot}}(\text{MeV})/300$.

Réponse à la question 2 :

```
double rigidite_mag(double Ecin, double Emass, double beta) {
// On suppose que Ecin et Emass soient specifiees en MeV
  double Etot = Ecin + Emass;
  return beta * Etot / 300;
}
```

Pour la déflexion d'un faisceau, il est possible d'utiliser des éléments électrostatiques ou magnétiques. La faisabilité et le prix doivent être pris en compte. Cela va dépendre du type de particules et de l'énergie du faisceau.
Si $|E| < 10^7$ V/m, un champ électrostatique sera utilisé.
Sinon, si $B < 1,8$ T, un champ magnétique à température ambiante sera choisi.
Sinon, un champ magnétique supraconducteur sera utilisé.

Question 3 : (1 point)

Écrire une fonction avec pour prototype : `void champ(double Brho, double rho, double beta)` qui affiche à l'écran quel type de champ est approprié dans notre cas. On rappelle que le champ électrique vaut $E = v\,B$.

Réponse à la question 3 :

```
void champ(double Brho, double rho, double beta) {
  double B, E, c = 3e8;
  B = Brho / rho;
  E = beta * c * B;
  if (E < 1e7)
    cout << "Champ electrostatique" << endl;
  else if (B < 1.8)
    cout << "Champ magnetique a temperature ambiante" << endl;
  else
    cout << "Champ magnetique supraconducteur" << endl;
}
```

Lorsque les particules d'un faisceau passent à travers un déflecteur, leurs coordonnées x et x' dans le plan transverse horizontal subissent la transformation $(x, x') \to (\bar{x}, \bar{x}')$ suivante, où $k = \sqrt{\frac{2-\beta^2}{\rho^2}}$ et $L = \rho\,\theta$:

$$\bar{x} = \cos(kL) \cdot x + \frac{1}{k}\sin(kL) \cdot x' \tag{2}$$

$$\bar{x}' = -k\sin(kL)\cdot x + \cos(kL)\cdot x' \tag{3}$$

Question 4 : (1 point)

Ecrire deux fonctions nommées `transport_x` et `transport_xp` qui calculent respectivement $\bar{x}$ et $\bar{x}'$.

Réponse à la question 4 :

```
double transport_x(double x, double xp, double beta, double rho, double theta) {
  double k, L;
  k = sqrt((2-beta*beta)/(rho*rho));
  L = rho*theta;
  return cos(k*L)*x + sin(k*L)*xp/k;
}
double transport_xp(double x, double xp, double beta, double rho, double theta) {
  double k, L;
  k = sqrt((2-beta*beta)/(rho*rho));
  L = rho*theta;
  return -k*sin(k*L)*x + cos(k*L)*xp;
}
```

Question 5 : (1.5 point)

Ecrire le programme principal `int main()` qui (en faisant appel aux fonctions des questions précédentes) fait afficher à l'écran le type de champ nécessaire pour le faisceau de rubidium et qui calcule les nouvelles coordonnées $\bar{x}$ et $\bar{x}'$ des $N = 5$ particules passant dans le déflecteur. Le programme doit également écrire ces nouvelles coordonnées dans un fichier nommé *coordonnees.res*.
Pour faire cela, vous pouvez supposer que la déclaration, l'allocation et l'initialisation (avec les valeurs initiales de x et x') de deux tableaux, dont les éléments sont `x[i]` et `xp[i]`, sont déjà faites ; ces lignes de code ne vous sont pas demandées.

Réponse à la question 5 :

```
int main() {
  double Ecin = 30.e-3, Emass = 85 * 931.5;
  double rot = 45*M_PI/180, rho = 0.2;
  double beta, Brho, temp;
  int i, N = 5;
  fstream fich;
// lignes pour declarer, allouer et initialiser x et xp pas demandees
  beta = beta_relat(Ecin, Emass);
  Brho = rigidite_mag(Ecin, Emass, beta);
  champ(Brho, rho, beta);
  fich.open("coordonnees.res", ios::out);
  for (i = 0; i < N; i++) {
    temp = x[i];
    x[i] = transport_x(x[i], xp[i], beta, rho, rot);
    xp[i] = transport_xp(temp, xp[i], beta, rho, rot);
    fich << x[i] << " " << xp[i] << endl;
  }
  fich.close();
  return 0;
}
```

2. Pointeurs

Question 1 : (0.75 point)

Expliquer quelle est l'utilité d'un pointeur sur une fonction.

Réponse à la question 1 :

Soit une fonction `A` qui fait appel à une autre fonction. Un pointeur sur une fonction comme argument de la fonction `A` permet de specifier au moment de son appel quelle fonction elle doit utiliser. Bien sûr, si la fonction `A` doit toujours utiliser la même fonction, on n'a pas besoin du pointeur.

Question 2 : (0.5 point)

Soit une fonction `galaxie` qui prend comme argument un entier nommé `systeme` et qui ne retourne aucune valeur. Écrire le prototype de cette fonction ainsi que le prototype d'une autre fonction qui la prend comme argument.

Réponse à la question 2 :

```
void galaxie(int systeme);
void univers(void (*g)(int));
```

Question 3 : (1 point)

Soit un tableau `int* p` à un indice de 9 éléments qui a été alloué et initialisé avec les valeurs suivantes :
8, 23, 34, 45, 56, 67, 78, 89, 90.
Quelles valeurs seront fournies par les syntaxes ci-dessous :
a) `p[6]` b) `*p+2` c) `*(p+2)` d) `&p+1` e) `(*&p)[p[2/3]]`

Réponse à la question 3 :

a) 78 b) 10 c) 34 e) 90
d) Une adresse de mémoire (c'est en fait l'adresse de `p` plus 8 octets, sur un système d'exploitation à 64 bits).

Soit le programme suivant, où il manque les opérateurs `*` et `&` :

```
 1. #include<iostream>
 2. #include<stdlib.h>
 3. using namespace std;
 4. void trier(int a, int b) {
 5.   int c;
 6.   if (a > b) {
 7.     c = a;
 8.     a = b;
 9.     b = c;
10.   }
11. }
12. void declar(int t, int n) {
13.   int i;
14.   t = (int)malloc(n*sizeof(int));
15.   for(i = 0; i < n; i++)
16.     t[i] = i;
17. }
18. void calcul(int a, int b, int t, int r) {
19.   int i;
20.   r = a / b;
21.   for(i = 0; i < b; i++)
22.     r += t[i];
23. }
24. int main() {
25.   int a = 5, b = 2, r;
26.   trier(a, b);
27.   int t;
28.   declar(t, b);
29.   calcul(a, b, t, r);
30.   cout << r << endl;
31.   free(t);
32.   return 0;
33. }
```

Question 4 : (1.5 point)

Ajouter des opérateurs `*` et `&` aux bons endroits et en bon nombre pour créer un programme qui tourne sans messages d'erreur. Vous devez aussi ajouter des parenthèses autour des pointeurs si elles sont indispensables. Il est interdit de faire d'autres changements au programme. Si vous avez des fonctions qui ne font rien, c'est faux ! Pour faire cela, vous copiez sur votre feuille seulement les lignes qui sont à changer, avec leur numéro. Marquer clairement ce que vous avez ajouté dans ces lignes.

Réponse à la question 4 :

```
 4. void trier(int* a, int* b) {
 6.   if (*a > *b) {
 7.     c = *a;
 8.     *a = *b;
 9.     *b = c;
12. void declar(int** t, int n) {
14.   *t = (int*)malloc(n*sizeof(int));
16.     (*t)[i] = i;
18. void calcul(int a, int b, int* t, int* r) {
20.   *r = a / b;
22.     *r += t[i];
26.   trier(&a, &b);
27.   int* t;
28.   declar(&t, b);
29.   calcul(a, b, t, &r);
```

Question 5 : (0.5 point)

Après avoir fait les corrections de la question précédente, qu'est-ce qui sera affiché par la commande `cout` de la ligne 30 ?

Réponse à la question 5 :

10

3. Mouvement d'une fusée

L'objectif de cet exercice est de définir des fonctions qui permettent une intégration numérique des équations différentielles qui régissent le mouvement d'une fusée. Un moteur fusée crée une poussée $\vec{F_p}$ en éjectant de la matière, accélérant ainsi la fusée en sens opposé de l'éjection. L'énergie nécessaire est obtenue le plus souvent à travers une réaction chimique (combustion) entre le carburant et le comburant. L'énergie chimique est d'abord convertie en énergie thermique, convertie elle-même par la tuyère en énergie cinétique des produits de combustion.

La particularité de ce problème de mécanique, simple à première vue, réside dans la variation de la masse de la fusée. Le carburant représente en effet une grande fraction de la masse totale au départ et qui diminue au fur et à mesure lors du vol. On note $m(t)$ la masse totale de la fusée, avec $m(t=0) = m_f + m_c$ où m_f correspond à la masse de la structure, des moteurs et de la charge utile, et m_c, la masse totale du carburant avant la mise à feu. On suppose que les produits de combustion sont éjectés avec une vitesse fixe par rapport à la fusée, notée u_e.

On considère ici le mouvement à une dimension d'une fusée selon l'axe Oz, dont la mise à feu a lieu à $t = 0$ en $z = 0$. $\vec{\kappa}$ est le vecteur unitaire selon Oz, $z(t)$ et $v(t)$ dénotent respectivement la position et la vitesse de la fusée. On note μ le débit massique constant des produits de combustion et on suppose que ceux-ci sont éjectés avec une vitesse u_e par rapport à la fusée. *(Noter le signe moins dans la relation entre $\dot{m}$ et μ.)*

$$\vec{v}(t) = v(t)\vec{\kappa} \qquad v(t) = \dot{z} = \frac{dz}{dt}$$

$$\vec{u_e} = -u_e\vec{\kappa} \qquad \frac{dm}{dt} = \frac{dm_c}{dt} = \dot{m} = -\mu$$

En écrivant la conservation de l'impulsion totale, on peut montrer que l'éjection des produits de combustion produit une poussée $\vec{F_p} = \mu u_e \vec{\kappa}$.

On suppose que la fusée n'est soumise à aucune autre force que celle due à la poussée des moteurs (pas d'effet de l'attraction gravitationnelle de la terre), sauf dans la question 7.

Question 1 : (1 point)

Définir les paramètres de la fusée ($m_f = 300$ kg, $m_c = 1700$ kg, $u_e = 2000$ m/s, $\mu = 10$ kg/s) comme des variables globales.
Écrire une fonction C qui calcule la dérivée de l'état et qui a comme prototype :

```
void syst_fusee(double t, double* q, double* qp, double (*fluxmot)(double, double*))
```

Le tableau `q` représente l'état de la fusée à l'instant `t` avec composantes $m(t)$, $z(t)$ et $v(t)$, et le tableau `qp` est la dérivée de l'état. L'argument `fluxmot` est un pointeur de fonction. Celle-ci a comme premier argument le temps et comme deuxième l'état `q` et elle calcule et renvoie le débit massique du moteur de la fusée.

Réponse à la question 1 :

```
double mf = 300, mc = 1700, ue = 2000, mu = 10;
void syst_fusee(double t, double* q, double* qp, double (*fluxmot)(double, double*)) {
  double mumu = fluxmot(t, q);
  qp[0] = -mumu;
  qp[1] = q[2];
  qp[2] = mumu * ue / q[0];
  // <-- Ajout question 7
}
```

Question 2 : (0.25 point)

Écrire une fonction `double flux_fixe(double t, double* q)` qui renvoie un débit massique μ constant, tant qu'il reste du carburant, zéro sinon.

Réponse à la question 2 :

```
double flux_fixe(double t, double* q) {
  if (q[0] <= mf)
    return 0;
  else
    return mu;
}
```

Question 3 : (0.75 point)

Écrire la fonction :

```
double euler_step(double t, double dt, int n, double* q, double* qp,
                  void (*fsys)(double, double*, double*))
```

qui calcule et met à jour l'état du système selon la méthode d'Euler. La fonction est du type `double` et non pas `void` car elle doit renvoyer la nouvelle valeur du temps, après incrémentation. L'état est stocké dans `q`, `t` et `dt` représentent le temps et le pas en temps et `n` le nombre de variables d'état. L'argument `fsys` correspond à la fonction qui calcule la dérivée de l'état du système (voir question suivante).

Réponse à la question 3 :

```
double euler_step(double t, double dt, int n, double* q, double* qp,
                  void (*fsys)(double, double*, double*)) {
  int i;
  fsys(t, q, qp);
  for (i = 0; i < n; i++)
    q[i] = q[i] + qp[i]*dt;
  return t+dt;
}
```

Question 4 : (0.25 point)

Écrire une fonction `void syst_fixe(...)`, conforme à celle attendue par la fonction `euler_step` définie ci-dessus, qui calcule la dérivée de l'état de la fusée pour un débit massique fixe, en utilisant les fonctions `syst_fusee` et `flux_fixe`.

Réponse à la question 4 :

```
void syst_fixe(double t, double* q, double* qp) {
  syst_fusee(t, q, qp, flux_fixe);
}
```

Question 5 : (1.5 point)

Écrire le programme principal dans lequel vous devez allouer les tableaux `q, qp`, définir le pas de temps $dt = 0,01$ s, créer une boucle qui effectue l'intégration numérique, et ajouter toutes les autres commandes nécessaires pour créer un programme fonctionnant. La boucle doit s'arrêter après l'extinction du moteur de la fusée. Le programme doit afficher à l'écran le temps, la position, la vitesse, l'accélération et la masse de la fusée à intervalle régulier, toutes les 5 secondes.

Réponse à la question 5 :

```
int main() {
  int n = 3;
  double t = 0, dt = 0.01;
  int nnt = 0, prtmod = 500; // Impression toutes les 5 secondes = 500*0.01
  double* q = (double*)malloc(n*sizeof(double));
  double* qp = (double*)malloc(n*sizeof(double));
  // (1) <-- Ajout question 6
  q[0] = mf + mc;
  q[1] = 0; q[2] = 0;
  while (q[0] > mf) {
    t = euler_step(t, dt, n, q, qp, syst_fixe);
    nnt++;
    if (nnt % prtmod == 0)
      cout << "t=" << t << " s z(t)=" << q[1] << " m v(t)=" << q[2]
           << " m/s accel=" << qp[2] << " m/s^2 m(t)=" << q[0] << " kg" << endl;
    // (2) <-- Ajout question 6
  }
  // (3) <-- Ajout question 6
  free(q); free(qp);
  return 0;
}
```

Question 6 : (1 point)

Ajouter à votre programme les lignes nécessaires pour calculer l'énergie cinétique totale de toute la masse éjectée, ainsi que l'énergie cinétique de la fusée après l'extinction du moteur, et pour les afficher à l'écran. On notera que la vitesse des masses éjectées dans le référentiel *fixe* s'écrit $(v(t) - u_e)\vec{\kappa}$.

Réponse à la question 6 :

On rajoute à l'endroit (1) indiqué dans le programme précédent :

```
  double Ecin_ejecta = 0;
```

à l'endroit (2) :

```
Ecin_ejecta += -0.5 * qp[0]*dt * (q[2]-ue)*(q[2]-ue);
```

et à l'endroit (3) :

```
cout << "Energie cinetique fusee = " << 0.5*q[0]*q[2]*q[2] << endl;
cout << "Energie cinetique totale ejecta = " << Ecin_ejecta << endl;
```

Question 7 : (0.5 point)

Modifier la fonction `syst_fusee` afin de prendre en compte l'attraction terrestre. Pour faire cela, introduire la constante de la gravitation $G = 6,67 \times 10^{-11}$ m^3 kg^{-1} s^{-2}, la masse de la terre $M_T = 5,97 \times 10^{24}$ kg et son rayon $R_T = 6,38 \times 10^6$ m comme variables globales. Le point $z = 0$ se trouve à la surface de la terre.

Réponse à la question 7 :

On rajoute les variables globales suivantes à celles de la question 1 :

```
double G = 6.67e-11, MT = 5.97e24, RT = 6.38e6;
```

et ensuite on rajoute à l'endroit indiqué dans la fonction `syst_fusee` de la question 1 :

```
qp[2] -= G*MT / ((q[1]+RT)*(q[1]+RT));
```

4. Neutrons, nombres aléatoires et méthode de Monte-Carlo

On se propose de simuler le transport de neutrons créés par une source stationnaire isotrope dans un domaine 1D homogène $[x_{min}, x_{max}]$ avec la méthode de Monte-Carlo. Dans ce contexte, un neutron est correctement représenté par sa position x et sa direction de propagation $\mu \in [-1, 1]$. Ici il faut imaginer que les neutrons bougent dans un plan et qu'on regarde seulement la projection de ces mouvements sur l'axe x. La variable μ représente alors $\cos\theta$ où θ est l'angle avec l'axe x.

On suppose que la population de neutrons est faible par rapport à la densité atomique du milieu, de sorte que les neutrons n'interagissent pas entre eux, mais seulement avec les noyaux des atomes. En notant λ le libre parcours moyen d'un neutron dans le milieu, on définit la section efficace totale par $\Sigma_t = 1/\lambda$. Cette grandeur est reliée à la probabilité d'interaction d'un neutron par unité de distance parcourue, elle dépend de la composition du milieu. Dans cet exercice, on suppose que seulement deux types de réactions peuvent survenir : soit le neutron est absorbé par un noyau, soit le neutron est diffusé (*scattering* en anglais) et modifie sa direction de propagation. Les sections efficaces d'absorption et de diffusion sont respectivement notées Σ_a et Σ_s. La section efficace totale vaut $\Sigma_t = \Sigma_a + \Sigma_s$.

La méthode de Monte-Carlo repose sur la simulation de l'histoire d'un grand nombre de particules par des processus stochastiques bien choisis. Lors de cette simulation, on va calculer des scores dans des régions du domaine, qui seront ensuite moyennés sur le nombre de particules simulées. Ces scores correspondent à des quantités physiques macroscopiques d'intérêt, par exemple le flux neutronique ϕ (proportionnel à la densité de neutrons).

Question 1 : (0.25 point)

On se donne les valeurs suivantes pour notre problème : $x_{min} = 0$ cm, $x_{max} = 100$ cm, $\Sigma_a = 0,02$ cm^{-1}, $\Sigma_s = 0,98$ cm^{-1}. Déclarer et initialiser les constantes `xmax`, `xmin`, `sigma_a`, `sigma_s`, `sigma_t` comme variables globales.

Réponse à la question 1 :

```
double xmin = 0, xmax = 100, sigma_a = 0.02, sigma_s = 0.98;
double sigma_t = sigma_a + sigma_s;
```

Question 2 : (0.25 point)

Ecrire une fonction `double sample_uniform(double min, double max)` qui renvoie un nombre aléatoire suivant une loi uniforme entre `min` et `max`. On utilisera la fonction drand48().

Réponse à la question 2 :

```
double sample_uniform(double min, double max) {
  return min + (max - min) * drand48();
}
```

On considère que les neutrons naissent à une position tirée aléatoirement selon une loi normale centrée au milieu du domaine et d'écart-type $\sigma = 10$. Leur direction initiale est uniforme dans $[-1, 1]$.

Question 3 : (1 point)

Écrire une fonction `double sample_birth_position()` qui tire une position de naissance dans $[x_{min}, x_{max}]$. On se servira de la transformation de Box-Muller, qui prend deux variables aléatoires u_1 et u_2 suivant une loi uniforme sur $[0, 1]$ et crée la variable gaussienne $z = m + \sigma\sqrt{-2\ln u_1}\cos(2\pi u_2)$ où m est l'espérance de la loi normale. On fera attention à écarter les points tirés en dehors de l'intervalle $[x_{min}, x_{max}]$.

Réponse à la question 3 :

```
double sample_birth_position() {
  double m = (xmin + xmax) / 2;
  double sigma = 10;
  double pos, u1, u2;
  do {
    u1 = sample_uniform(0, 1);
    u2 = sample_uniform(0, 1);
    pos = m + sigma * sqrt(-2 * log(u1)) * cos(2 * M_PI * u2);
  } while (pos < xmin || pos > xmax);
  return pos;
}
```

Entre deux réactions, un neutron voyage en ligne droite suivant sa direction μ. La longueur du vol s (dans le plan) avant la prochaine réaction suit une loi exponentielle qui dépend de Σ_t, et peut être échantillonnée par

$$s = \frac{-\ln(1-u)}{\Sigma_t}, \tag{4}$$

où u suit une loi uniforme sur $[0, 1]$.

Question 4 : (0.5 point)

Écrire une fonction nommée `sample_new_pos` qui échantillonne une nouvelle position pour le neutron connaissant sa position de départ et sa direction.

Réponse à la question 4 :

```
double sample_new_pos(double pos, double dir) {
  return pos - dir * log(1 - sample_uniform(0, 1)) / sigma_t;
}
```

Tant qu'un neutron n'est pas absorbé, il vole en ligne droite jusqu'à une nouvelle position, où il réagit avec un noyau du milieu. Il a une probabilité $p_a = \Sigma_a/\Sigma_t$ de se faire absorber et une probabilité $p_s = \Sigma_s/\Sigma_t = 1 - p_a$ de se faire diffuser. Une absorption entraîne la mort du neutron. Lors d'une diffusion on échantillonne la nouvelle direction de vol selon une loi uniforme sur $[-1, 1]$, et on réitère le processus.

Question 5 : (1 point)

Écrire une fonction nommée `simulate_history` qui prend en arguments la position et la direction initiales d'un neutron et simule son histoire jusqu'à sa mort. Si le neutron sort du domaine après un vol, il est considéré comme mort.

Réponse à la question 5 :

```
void simulate_history(double pos, double dir) {
  double u;
  // (1) <-- Ajout question 8
  while (1) {
    pos = sample_new_pos(pos, dir);
    if (pos < xmin || pos > xmax)
      break;
    // (2) <-- Ajout question 8
    // <-- Ajout question 9
    u = sample_uniform(0, 1);
    if (u < sigma_a / sigma_t)
      break;
    else
      dir = sample_uniform(-1, 1);
  }
}
```

Question 6 : (1 point)

Écrire la fonction `main`, qui simule l'histoire de N particules. Le programme doit demander la valeur de N à l'utilisateur. On n'oubliera pas d'initialiser la graine du générateur aléatoire.

Réponse à la question 6 :

```
int main() {
  int i, N;
  double pos, dir;
  // (1) <-- Ajout question 8
  srand48(time(NULL));
  cout << "Entrez le nombre de particules à simuler : ";
  cin >> N;
  for (i = 0; i < N; i++) {
    pos = sample_birth_position();
    dir = sample_uniform(-1, 1);
    simulate_history(pos, dir);
  }
  // <-- Ajout question 9
  // (2) <-- Ajout question 8
  return 0;
}
```

On peut maintenant simuler le mouvement et les réactions des neutrons dans le domaine. Mais pour l'instant, ces simulations sont inutiles car on ne calcule rien ! Dans cette seconde partie, on va voir comment calculer le flux neutronique ϕ à partir de la simulation de particules individuelles.

Question 7 : (0.5 point)

On se donne $n_x = 100$ cellules sur $[x_{min}, x_{max}]$ avec une taille dx constante. Déclarer et initialiser les constantes `nx` et `dx` comme variables globales. Écrire une fonction nommée `find_cell` qui prend en argument une position dans le domaine et renvoie l'indice de la cellule associée.

Réponse à la question 7 :

```
int nx = 100;
double dx = (xmax - xmin) / nx;
int find_cell(double pos) {
  return (pos - xmin) / dx;
}
```

On se propose de calculer le flux neutronique. Pour cela, on va créer un tableau nommé `flux` à deux indices, de n_x lignes et 2 colonnes, initialement nul, qui va stocker la moyenne empirique du flux $\overline{\phi_i}$ dans chaque cellule dans sa première colonne (la deuxième colonne servira pour la question d'après). Lors de la simulation, à chaque fois qu'une particule subit une réaction dans la cellule i, on ajoute $1/\Sigma_t$ à $\overline{\phi_i}$, puis on divise par N à la fin de la simulation pour obtenir la moyenne.

Question 8 : (1.25 point)

Dans la fonction `main`, déclarer, allouer et initialiser le tableau `flux` à deux indices. Ecrire également les instructions pour libérer le tableau à la fin de la fonction. Ensuite faites les modifications nécessaires dans `main` et les autres fonctions pour calculer la moyenne du flux dans chaque cellule avec la méthode décrite ci-dessus.

Réponse à la question 8 :

On rajoute aux arguments de la fonction `simulate_history` de la question 5 un troisième argument : `double** flux`. Ensuite on rajoute à l'endroit (1) indiqué dans cette fonction :

```
int cell;
```

et à l'endroit (2) :

```
cell = find_cell(pos);
flux[cell][0] += 1 / sigma_t;
```

Puis on rajoute à l'endroit (1) indiqué dans le programme principal de la question 6 :

```
double** flux = (double**)malloc(nx * sizeof(double*));
for (i = 0; i < nx; i++) {
  flux[i] = (double*)malloc(2 * sizeof(double));
  flux[i][0] = 0;
  flux[i][1] = 0;
}
```

et à l'endroit (2) :

```
for (i = 0; i < nx; i++)
  flux[i][0] /= N;
// Bien sur, normalement on ferait quelque chose avec ce resultat ici avant de l'effacer,
// mais cela ne fait pas partie de l'exercice
for (i = 0; i < nx; i++) {
  free(flux[i]);
}
free(flux);
```

Enfin il ne faut pas oublier de rajouter l'argument `flux` à l'appel de la fonction `simulate_history`.

$\overline{\phi_i}$ est un estimateur entaché d'erreur, auquel on peut associer une variance V_i. Lors de chaque réaction dans la cellule i, on ajoute $(1/\Sigma_t)^2$ à V_i (initialisée à zéro), qu'on stockera dans la deuxième colonne du tableau `flux`. À la fin de la simulation, la variance dans chaque cellule i est calculée par

$$V_i \longleftarrow \frac{1}{N-1}\Big(\frac{V_i}{N} - \overline{\phi_i}^2\Big). \tag{5}$$

Question 9 : (0.5 point)

Modifier le programme pour calculer la variance en plus de la moyenne, selon la méthode décrite ci-dessus.

Réponse à la question 9 :

On rajoute à l'endroit indiqué dans la fonction `simulate_history` de la question 5 :

```
flux[cell][1] += 1 / (sigma_t * sigma_t);
```

On rajoute à l'endroit indiqué dans le programme principal de la question 6 :

```
for (i = 0; i < nx; i++) {
  flux[i][1] = (flux[i][1] / N - flux[i][0] * flux[i][0]) / (N - 1);
}
```