TP 3

Adsorption de particules sur une surface

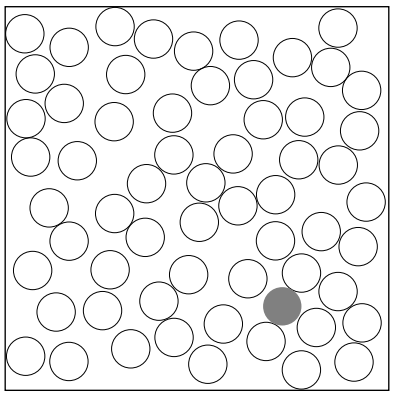
3.1 Surface homogène

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.

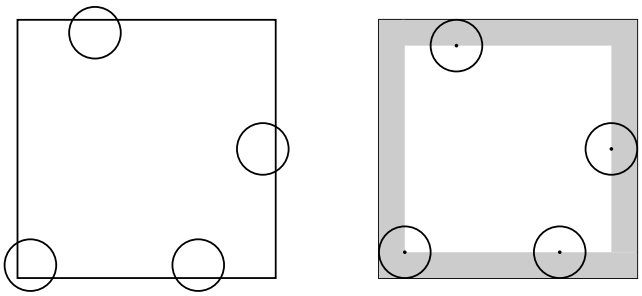
Pour modéliser ce problème, on fait les hypothèses préliminaires suivantes :

- 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

```
const double L = 20.0;
const double R = 0.4;
const int  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 C/C++ 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 soit on utilise deux tableaux, un pour x et un pour y, soit on assemble ces deux valeurs réelles dans un "struct" du C ou encore une "classe" du C++, puis on fait un seul tableau de ce nouveau type "coordonnes". Pour en savoir plus sur les "struct" et les "class", voir le poly de cours C++ 3P002, chapitre 14.2 "struct du C" et chapitre 14.5 "Exemple : TP5 avec C++".

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.

On vous propose ici deux manières alternatives pour enregistrer tous les cercles de la surface :

1. Avec la classe vector de la STL (standard template library)
2. Avec les tableaux classiques du C

La première variante avec "vector" a l'avantage qu'elle implémente déjà le tableau dynamique, vous n'avez pas besoin de l'implémenter par vous même. Puis cela peut éviter des bugs de débordement ou mauvaise indexation d'un tableau classique en C.

La deuxième variante a l'avantage d'utiliser ce que vous connaissez déjà : les tableaux classiques en C. Comme ces tableaux ne sont pas dynamiques - on ne peut pas changer leur taille lors de l'exécution - il faudra par contre implémenter par vous même cette fonctionnalité de tableau dynamique sans faire d'erreur au niveau des indices des tableaux.

Les deux variantes sont aussi rapides au niveau de l'exécution du programme, si on respecte les consignes données ci-dessous.

Voir plus bas pour savoir comment faire, puis à vous de choisir ce que vous préférez.

L'enregistrement des cercles avec la classe vector de la STL La classe "vector" de la STL (standard template library) est introduite dans les slides `coursTD4_2020.pdf` sur moodle (dossier Ressources). Ici est rappelé tout dont on a besoin pour ce TP, si on choisit d'utiliser "vector" pour enregistrer les cercles.

Pour commencer il faut importer la classe vector : **#include** <vector>. Comme la STL est fournit avec chaque compilateur C++, il n'y a pas besoin de l'installer pour utiliser la classe vector.

Ensuite, dans la fonction gérant l'évolution de notre système, on crée un tableau vide (ici pour l'exemple de la coordonné x) : vector<**double**> x; Comme la classe vector permet de stocker des variables et des objets de n'importe quel type, il faut indiquer ici le type des éléments du vector via <**double**>. Si vous utilisez par exemple la classe "Coordinate" du chapitre 14.5 du poly de cours C++ 3P002, alors il aurait fallu écrire : vector<Coordinate> c.

A chaque fois qu'un cercle est adsorbé sur la surface, on ajoute ces coordonnées `x_new` et `y_new` à la fin du vector avec la fonction push_back() : x.push_back(x_new). Cette fonction agrandit le tableau vector d'une case et copie&colle la valeur fournit (ici `x_new`) dans cette nouvelle case.

Le nombre de cercles enregistrés est donné par la fonction size() de la classe vector : x.size(). Pour parcourir les éléments d'un vector, c'est comme pour un tableau classique en C, sauf que la taille donnée par x.size() est toujours correcte :

```
for (int i=0; i < x.size(); i++){
    cout << x[i] << endl;
}
```

Alternativement on peut faire aussi ceci pour lire tous les éléments d'un vector :

```
for (auto elem: x){
```

```
    cout << elem << endl;
}
```

Pour ceux qui ont déjà fait du python, cette notation du C++ moderne (C++11) est l'équivalent du "for elem in x :".

Si vous souhaitez passer le vector à une autre fonction `func`, alors le mieux serait d'utiliser une "référence", indiqué avec le symbole & dans l'entête de la fonction : **void** func(vector<**double**>& v). Cela évite que toutes les cases du vector soient copiées&collées à chaque appel de la fonction. Voir la partie 12.3.2 "Passer une matrice à une autre fonction" du poly de cours C++ 3P002.

Pour optimiser le temps d'exécution, on peut réserver à l'avance suffisamment de cases dans la mémoire vive avec la fonction reserve(), ici alors x.reserve(N_MAX). Cela ne change pas la taille du vector donnée par size(), donc le vector reste un tableau dynamique. Puis lors de la compilation l'ajout de l'option `-O2` (grand O, comme Optimisation 2ème niveau) permet ici de gagner un facteur 10 à 20 environ en vitesse d'exécution.

L'enregistrement des cercles avec les tableaux classiques du C Ici le tableau dynamique est implémenté avec un tableau statique du C de taille `N_MAX` et grâce à l'utilisation d'une variable `n_at` qui représente le nombre de cercles déjà présents dans le système (l'équivalent du `x.size()` de la classe vector).

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

```
int n_at = 0;          // nombre de cercles déjà présents dans le système
double x[N_MAX]; // x[i] est l'abscisse du i-ème cercle présent
double y[N_MAX]; // y[i] est l'ordonnée du i-ème cercle présent
```

Ces variables seront ensuite transmises par argument (pointeurs) 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 $0 \leq i < n\_at$, donc pour parcourir les coordonnées des cercles déjà adsorbés, on fait :

```
for (int i=0; i < n_at; i++){
    cout << x[i] << "  " << y[i] << endl;
}
```

Au départ on a `n_at␣=␣0`, c'est cette variable qui détermine l'emplacement des deux tableaux où on doit ajouter un nouveau cercle : x[n_at] = x_new. Après chaque ajout `n_at` est incrémenté de un. Par construction `n_at` doit toujours rester bien inférieur à `N_MAX`, sinon il s'agit d'un bug. Pour vider la surface il suffit de faire `n_at=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 **double** coord(**void**) 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 **double** coord(**void**) 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() deux fois.
- Une fonction **int** remplissage(**int** MAX_TRIES) qui gère l'évolution du système pour arrive 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 :

```
int remplissage(int MAX_TRIES){
    ... // déclaration tableaux C ou vector pour enregistrer les cercles
    int echecs = 0;
    while(echecs < MAX_TRIES){
        double x_new = coord(); double y_new = coord(); // tirage aléatoire de coordonnées
        // Test si l'emplacement est libre :
        int libre = 1; // supposer au départ que oui
        for (...) // parcourir tous les cercles déjà adsorbés
            if (...) // pour voir s'il y a un qui chevauche
                libre = 0; // si oui, alors la place n'est pas libre
        // Ajout de cercle si emplacement libre, sinon incrémenter echecs
        if ( libre == 1)
            ... // ajout de cercle dans les tableaux C ou vector
            echecs = 0; // remettre le compteur à zéro pour le prochain cercle
        else
            echecs++;
    }
    return ... // retourner le nombre de cercles adsorbés
}
```

3.1.2 Analyse des résultats

2. Utilisez gnuplot pour visualiser la configuration finale. Pour cela modifier votre programme afin qu'il enregistre un fichier avec trois colonnes "x y R" et une ligne par cercle :

1. x : les positions en x des cercles
2. y : les positions en y des cercles
3. R : rayons des cercles (ici toujours la même valeur pour tous les cercles)

Sous gnuplot il faudra tracer ces données de cette façon :

```
set size square
plot 'fichier.res' with circles
```

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 votre programme fonctionne bien, enlever pour la suite les lignes de code qui servent à enregistrer un fichier pour gnuplot.

3. Modifier la fonction main() pour que l'expérience soit faite `M` = 1000 fois et que la fonction main() affiche 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.

Pour vérifier si votre programme fournit un bon résultat, voici les valeurs qu'on devrait obtenir approximativement :

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

4. Comment se comparent les valeurs obtenues à la fraction qu'on pourrait idéalement occuper d'une façon ordonnée ?

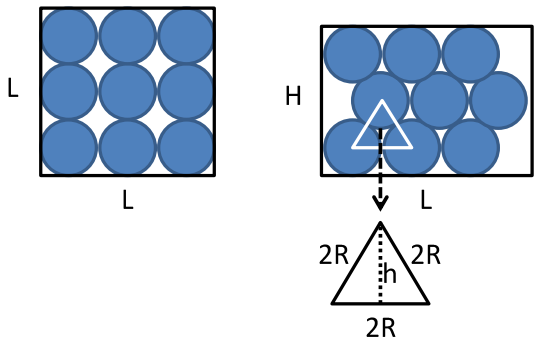


FIGURE 3.2 – Empilement ordonné, *à gauche :* empilement carré, *à droite :* empilement hexagonal

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 :

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 :

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

Voir en annexe 3.4.1 de ce TP pour de la lecture intéressante sur l'empilement de bobines ou encore des bonbons M&Ms !

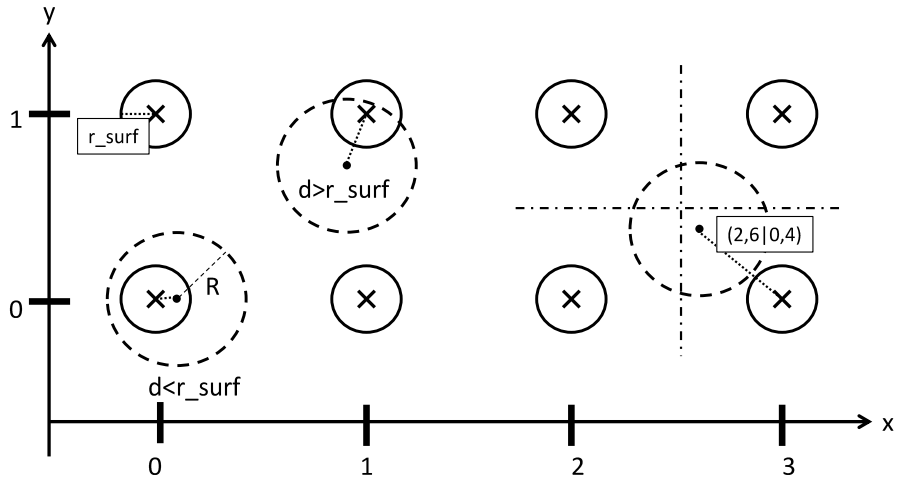


FIGURE 3.4 – Modèle d'adsorption sur un réseau d'atomes, ici en cercles pleins avec croix. Les particules de gaz, ici en cercles avec tirets, peuvent soit s'adsorber dans tous les cas, si $d <$ r_surf ou alors seulement avec une certaine probabilité, si $d >$ r_surf. Comme les atomes ont des coordonnées 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.

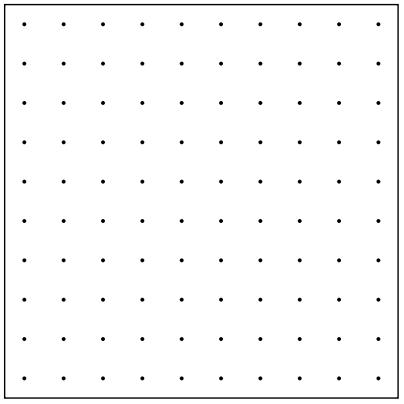


FIGURE 3.3 – Surface d'adsorption avec sa structure atomique, les atomes étant organisés en réseau carré.

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,

```
const double r_surf = 0.05;
const double U = 10.0;
```

Il faudra

- Définir une fonction **double** dist_latt(**double** x_new, **double** 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 **double** rint(**double**) de la librairie mathématique 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 (..., **double** T) et doit implémenter l'algorithme décrit.
- Modifier la fonction main() 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 notre fonction `alea()` du TP0 qui fournit un nombre (pseudo-)aléatoire entre zéro et un d'une manière uniforme. La probabilité que la condition `alea() < 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 `alea() < 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(alea()) < -U`

3.2.2 Dépendance de la température

7. Comme avant utiliser gnuplot 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). 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`.

9. En ayant analysé les configurations avec gnuplot (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 :* Influence de MAX_TRIES sur les résultats

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=100` répétitions seulement pour limiter le temps de calcul. Tracer la fraction moyenne de surface occupée en fonction de MAX_TRIES en échelle semi-log.

Commentez le graphe.

11. Essayer d'étendre la courbe jusqu'à `MAX_TRIES` = 512.000 et avec `M=100` répétitions. Pour cela n'oubliez pas de compiler votre programme C++ avec l'option "-O2" (O comme Optimisation). Commentez le nouveau graphe.

3.4 Annexe

3.4.1 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.