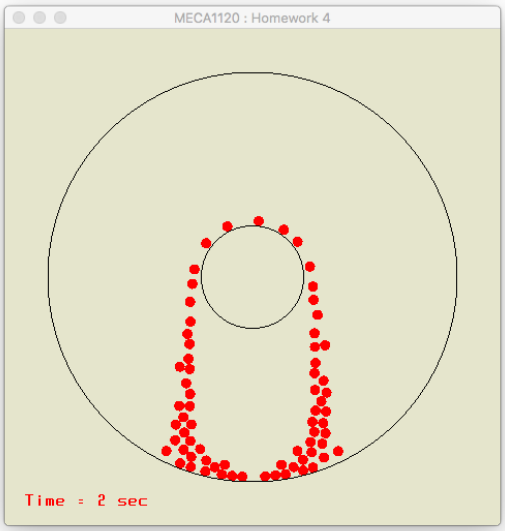
Finite elements for dummies : Des grains en folie :-)



Nous allons maintenant écrire un solveur de contact pour des éléments discrets. Il s'agira de calculer des corrections de vitesses pour tenir compte des impulsions que vont ressentir des billes lorsqu'elles entrent en contact entre elles ou avec les deux frontières extérieures du domaine. On considère n billes qui sont soumises à la gravité et à une force de trainée linéaire.

$$m_i \frac{\Delta \mathbf{v}_i}{\Delta t} = m_i \mathbf{g} - \gamma \mathbf{v}_i$$

Pour adapter la position des billes, il suffit d'écrire :

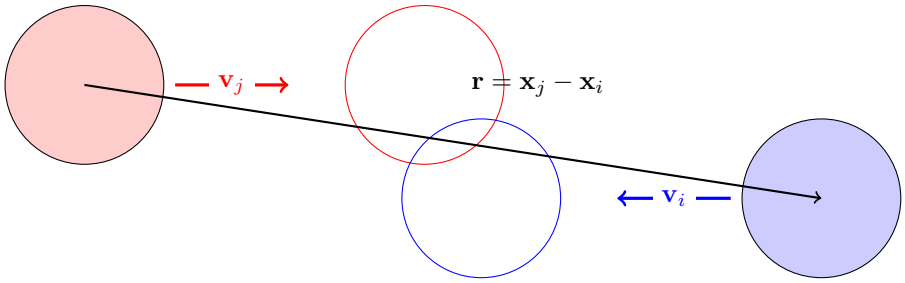
$$m_i \frac{\Delta \mathbf{x}_i}{\Delta t} = \mathbf{v}_i$$

Malencontreusement de cette manière, on ne tient pas compte des collisions qui se sont effectuées entre les billes et avec les deux frontières. Il faut donc de corriger les vitesses afin d'éviter toute inter-pénétration : c'est un problème compliqué et largement non-linéaire. On doit donc procéder de manière itérative. Lors d'une itération, on traite chaque collision possible de manière séquentielle en incluant éventuellement une correction de vitesse pour les grains concernés. En outre, on conserve l'ensemble des corrections requises pour chaque contact possible.

Mais, cette première ébauche de correction n'est souvent pas adéquate, car une collision peut avoir un impact sur une autre collision possible... Le calcul des corrections est donc couplé pour tous les grains. Il faut vérifier que la solution proposée est compatible. Pour chaque collision observée, on retire la correction suggérée aux vitesses (c'est pourquoi, il est nécessaire de stocker ces corrections !) et on vérifie si il est toujours nécessaire d'appliquer une correction ou si la correction proposée est adéquate. Si une correction n'est plus utile, on supprime alors la correction proposée qu'on retranche des vitesses modifiées. Sinon, on recalcule la correction requise et on la compare avec la valeur précédente. C'est la valeur maximale de la valeur absolues de tous les incréments de corrections locales qui permet d'estimer l'erreur ζ de la solution obtenue dans le processus itératif qui va converger de manière assez lente au passage...

Lorsque le schéma itératif a convergé, on peut montrer que ces corrections de vitesses permettront d'obtenir les forces qui agissent sur les grains lorsqu'un contact est observé entre les grains.... Le solveur de contact fournit les vitesses qui permettent d'éviter toute inter-pénétration, mais également les forces qui apparaissent pour les billes en contact entre elles ou avec la frontière. Ainsi, de manière assez paradoxale, le problème de contact est particulièrement difficile lorsque toutes les billes se trouvent à l'équilibre en bas. Heureusement, on utilise la solution obtenue au pas de temps précédent pour éviter de reproduire un long processus itératif lorsque plus rien ne bouge ! Plus précisément, la description précise de l'algorithme du solveur de contact est décrit comme suit :

Calcul d'une correction de vitesse pour tenir compte des collisions entre les grains.



Pour chaque contact, on calcule γ la distance entre les deux grains et on estime la correction de vitesse normale relative qu'il faudrait retrancher globalement aux grains si ceux risquaient de se recouvrir après un instant Δt.

$$\begin{aligned} \gamma &= r - (r_i + r_j) \\ v_n &= \mathbf{v}_i \cdot \mathbf{n} - \mathbf{v}_j \cdot \mathbf{n} \\ \delta v &= \max(0, v_n + \delta v_c - \gamma/\Delta t) - \delta v_c \end{aligned}$$

où $\mathbf{n}$ est la normale entre les deux grains que l'on obtient en normalisant le vecteur $\mathbf{r}$. Ensuite, on retranche les corrections à chaque grain, en supposant une collision inélastique : on conserve donc la quantité de mouvement globale. On met à jour δv_c la correction associée au contact et le critère global d'erreur ζ.

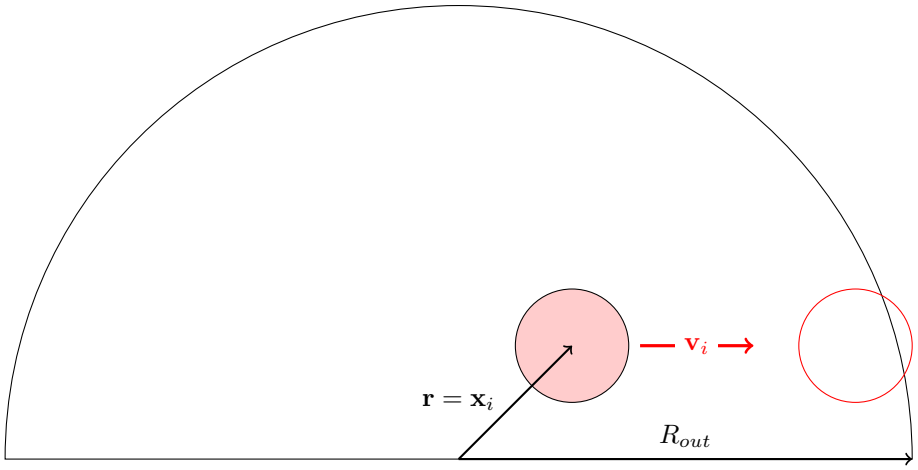
$$\Delta\mathbf{v}_i = -\delta v\ \mathbf{n}\ \frac{m_j}{m_i + m_j} \qquad \Delta\mathbf{v}_j = \delta v\ \mathbf{n}\ \frac{m_i}{m_i + m_j}$$

$$\Delta\delta v_c = \delta v \qquad \zeta = \max(\zeta, \mathrm{abs}(\delta v))$$

Intuitivement, on voit qu'on applique davantage la correction sur la bille de masse moindre que sur l'autre.... C'est donc bien le plus gros qui impose sa loi :-)

Attention, pour la toute première itération, on se contente d'uniquement mettre à jour les vitesses sans rien faire d'autre :-) Cela permet d'utiliser les corrections obtenues au pas de temps précédente, ce qui fournit, en général, un excellent candidat initial. Il faut uniquement effectuer la correction de vitesses, afin que les vitesses et les corrections soient compatibles pour l'itération suivante.

Calcul d'une correction de vitesse pour tenir compte des collisions avec la frontière.



Pour chaque grain, on calcule les distances aux deux frontières et on estime la correction de vitesse normale relative qu'il faudrait retrancher au grain si il risquait de franchir l'une des deux frontières après un instant Δt.

$$
\begin{aligned}
v_n &= \mathbf{v}_i \cdot \mathbf{n} \\
\delta v &= \max(0, v_n + \delta v_b - \underbrace{(R_{out} - r - r_i)}_{\gamma_{out}}/\Delta t, -v_n - \delta v_b - \underbrace{(r - R_{in} - r_i)}_{\gamma_{in}}/\Delta t) - \delta v_b
\end{aligned}
$$

où $\mathbf{n}$ est la normale entre le grain et la frontière que l'on obtient en normalisant le vecteur $\mathbf{r}$. Ensuite, on retranche les corrections à chaque grain. On met à jour δv_c la correction associée au contact et le critère global d'erreur ζ.

$$
\Delta \mathbf{v}_i = -\delta v \ \mathbf{n} \qquad \Delta \delta v_b = \delta v \qquad \zeta = \max(\zeta, \mathrm{abs}(\delta v))
$$

Pour la toute première itération, on se contente à nouveau de mettre à jour les vitesses :-)

Et pratiquement...

La totalité des données et des fonctions que nous aurons besoin pour résoudre ce problème sont :

```
typedef struct {
    int n;
    double radiusIn;
    double radiusOut;
    double gravity[2];
    double gamma;
    double *x;
    double *y;
    double *vx;
    double *vy;
    double *r;
    double *m;
```

```
    double *dvBoundary;
    double *dvContacts;
} femGrains;


femGrains  *femGrainsCreateSimple(int n, double r, double m, double radiusIn, double radiusOut);
void        femGrainsFree(femGrains *myGrains);
void        femGrainsUpdate(femGrains *myGrains, double dt, double tol, double iterMax);
double      femGrainsContactIterate(femGrains *myGrains, double dt, int iter);
```

Plus précisément, on vous demande de concevoir, d'écrire ou de modifier deux fonctions.

1. Tout d'abord, il faut compléter la fonction `femGrainsUpdate` qui met à jour les positions et les vitesses pour un pas de temps `dt`. Cette fonction effectue trois étapes à chaque pas de temps.
   - Elle met à jour, les vitesses sur base de la loi de Newton : il vous est demandé d'écrire cette étape assez élémentaire qui mettra les grains en mouvement...
   - Ensuite, elle lance un processus itératif pour calculer les corrections des vitesses afin de tenir compte des collisions. Chaque itération du processus itératif sera réalisée dans la fonction `femGrainsContactIterate`. Cette dernière fonction fournit en retour une estimation de l'erreur commise lors de chaque itération, c'est la valeur `zeta`. On observe que le processus itératif de résolution des contacts sera arrêté lorsque cette erreur est inférieure à la tolérance requise ou que le nombre maximal d'itérations est atteint. On observera que le processus de convergence du solveur de contact n'est pas particulièrement rapide.
   - Finalement, on corrige les positions sur base de vitesses corrigées obtenue lors de la seconde étape. Cette partie du code vous est gracieusement offerte par l'équipe didactique !
2. Ensuite, vous vous attaquez à l'écriture de la fonction `femGrainsContactIterate` qui effectue une itération du solveur de contact. Pour effectuer ce calcul, on conserve toutes les corrections requises pour chaque contact entre les billes dans le tableau `dvContacts[nContacts]` et les corrections requises pour les collisions avec une des deux frontières dans `dvBoundary[n]` avec `nContacts = n(n-1)/2`.
3. Il est possible d'interrompre de relancer la simulation avec les touche `R` et `S`. Il est aussi possible de dé-commenter quelques lignes de code dans le programme principal pour contrôler la progression de chaque itération temporelle en introduisant un caractère dans la fenêtre de commande.
4. Vos deux fonctions seront incluses dans un unique fichier `homework.c`, sans y adjoindre le programme de test fourni ! Ce fichier devra être soumis via le web et la correction sera effectuée automatiquement. Il est donc indispensable de respecter strictement la signature des fonctions. Votre code devra être strictement conforme au langage `C` et il est fortement conseillé de bien vérifier que la compilation s'exécute correctement sur le serveur.
5. Afin de pouvoir effectuer un test distinct de vos fonctions, des instructions de compilation ont été mis dans le fichier `homework.c`. Il est IMPERATIF de ne pas les retirer, ni de les modifier...