Rendu de TIPE : Simulation physique et fluide

Etienne THOMAS, Alban COADIC

17 juin 2025

Table des matières

1 Préambule

On cherche à modéliser l'effet du vent sur un bâtiment afin d'observer les vortexes de Von Kármán. On simule alors une structure de points, reliés par des liens indéformables (n-pendule, ponts, etc.) dans un fluide en mouvement, La simulation de fluide est traité dans le TIPE d'Alban COADIC et la simulation physique dans celle d'Etienne THOMAS. À l'heure actuelle, les simulations sont en 2D, dans un espace fini. La simulation est capable de récupérer les forces appliquées sur les liens et donc les potentiels points de rupture.
Certaines améliorations sont envisagées :
— Passer a une simulation en 3D
— introduire la mecanique du solide
— Régler les problèmes de conditions aux limites de la simulation de fluide
— Passer à un univers infini
— Mettre en place une méthode d'adaptative mesh refinement

2 Introduction

Les simulations sont discrétisées car celle-ci corresponde des systèmes d'ODLE sans solutions analytique. À chaque pas de temps (pas), le programme doit faire évoluer le système en respectant les lois suivantes :
— Les lois **Newton** pour la partie physique :
  — Si $\sum \overrightarrow{F_{ext}} = \overrightarrow{0}$, alors le mouvement du corps est rectiligne uniforme, c'est-à-dire $\frac{d\overrightarrow{v}}{dt} = \overrightarrow{0}$
  — $\overrightarrow{p} = \sum \overrightarrow{F_{ext}}$
  — À toute action correspond une réaction égale et opposée

— Les contraintes géométriques entre les points (les liens).
— Les lois de **Navier-Stokes** pour la partie fluide :
  — Conservation de la masse (incompressibilité (divergence)) :

  — Conservation de la quantité de mouvement :

$$\underbrace{\frac{\partial \vec{v}}{\partial t}}_{\substack{\text{variation temporelle} \\ \text{de la vitesse}}} = \underbrace{-\nabla p}_{\substack{\text{gradient de pression} \\ \text{(force due à la pression)}}} + \underbrace{\rho \vec{g}}_{\substack{\text{force de gravité} \\ (\rho\text{=masse volumique,} \\ \vec{g}\text{=gravité})}} + \underbrace{\mu \nabla^2 \vec{v}}_{\substack{\text{viscosité} \\ (\mu\text{=viscosité dynamique})}}$$

3 Simulation physique

3.1 Introduction

La difficulté de la simulation physique est de maintenir en même temps les lois de Newton et les contraintes géométriques.

Exemple : Exemple d'un lien rigide entre deux masses :

$$C : ||\vec{q}|| - l = 0$$

Notation : $\vec{q}$ le vecteur position de la masse, l la longueur de la tige

3.2 Définition

Pour maintenir les contraintes, on cherche a trouver une expression de la force de correction F_c à appliquer sur les masses, qui permet de maintenir la contrainte C. Afin d'obtenir une contrainte sur la vitesse puis l'acceleration, on dérive la contrainte par rapport au temps :

$$\dot{C} : \frac{dC}{dt} = \frac{\partial C}{\partial q}\dot{q}$$

Notation :

J $= \frac{\partial C}{\partial q}$, ainsi $\dot{C} : J\dot{q} = 0$. J est la matrice jacobienne de la fonction associée à la contrainte

$$\ddot{C} : \dot{J}\dot{q} + J\ddot{q} = 0$$

$\dot{J}\dot{q}$ est une écriture inutilement complexe, pour obtenir ce terme, il est plus facile de passer par l'égalité suivante

Exemple : Pour un pendule, C est une fonction de deux variables, sa jacobienne est une matrice 1×2, $(J = \frac{q^\intercal}{||q||})$

Par définition, les contraintes géométriques de doivent pas introduire de l'énergie dans le système : elles ne travaillent pas. On en déduit la forme que les force de correction doivent prendre :

$$\forall \lambda \in \mathbb{R}, (J^\intercal \lambda) \cdot \dot{q} = (J^\intercal \lambda)^\intercal \dot{q} = \lambda J \dot{q} = 0$$

L'inconnue recherchée dans la suite du problème est alors le coefficient λ.

3.3 Changement d'ordre

Résoudre sur la position (ordre 0) est un problème non linéaire et travailler sur la vitesse (ordre 1) est plus simple.
Le programme exécute dans l'ordre les étapes suivantes :

1. Intégrer la vitesse, via la méthode d'Euler implicite

$$\underline{\dot{q}_{n+1}} = \dot{q}_n + \Delta t W F$$

2. Corriger la vitesse
3. Intégrer la position à partir de la vitesse précédemment corrigée pour obtenir une contrainte sur la vitesse, on applique la règle de la chaîne :

$$\dot{C} : \frac{dC}{dt} = \frac{\partial C}{\partial q}\dot{q}$$

Notation : $J = \frac{\partial C}{\partial q}$, ainsi $\dot{C} : J\dot{q} = 0$. J est la matrice jacobienne de la fonction associée à la contrainte

Exemple : Pour un pendule, C est une fonction de deux variables, sa jacobienne est une matrice 1×2, $(J = \frac{q^\intercal}{||q||})$

3.4 Definition de l'inconnue

Les forces de corrections recherchées ne travaillent pas : $\mathcal{P}(F_c) = F_c \cdot \dot{q}$.
Remarquons que :

$$\forall \lambda \in \mathbb{R}, (J^\intercal \lambda) \cdot \dot{q} = (J^\intercal \lambda)^\intercal \dot{q} = \lambda J\dot{q} = 0$$

Tout force colinéaire à J ne travaille pas, par conséquent, on cherche à isoler λ pour trouver la force de correction.

Notation : λ est le coefficient Lagrangien

3.5 Mise en équation

Notation : $\underline{\dot{q}}$ est la vitesse avant correction, $\dot{q_c}$ est la correction recherchée et P_c est la quantité de mouvement simplement

$$\begin{aligned}
yy\dot{C} &\Leftrightarrow J\dot{q} = 0 \\
&\Leftrightarrow J(\dot{q_c} + \underline{\dot{q}}) = 0 \\
&\Leftrightarrow J\dot{q_c} = -J\underline{\dot{q}} \\
&\Leftrightarrow JWP_c = -J\underline{\dot{q}} \\
&\Leftrightarrow JW\Delta t F_c = -J\underline{\dot{q}} \\
&\Leftrightarrow JW\Delta t J^\intercal \lambda = -J\underline{\dot{q}} \\
&\Leftrightarrow JWJ^\intercal \lambda' = -J\underline{\dot{q}}
\end{aligned}$$

En posant $\lambda' = \Delta t\lambda$, nous avons ainsi $P_c = J^\intercal \lambda'$.

Remarque : $JWJ^\intercal$ peut être interprété comme l'inverse de la masse perçu par la contrainte.

Exemple : Dans un système avec une unique contrainte, $JWJ^\intercal$ est une matrice 1×1, on se ramène donc à équation polynomiale du premier degré

3.6 Extension a un système de plusieurs contraintes

Pour étendre la méthode à n objets et p contraintes, il faut considérer l'ensemble des vecteurs position comme un unique vecteur q de dimension $n \cdot d$,

Notation : d est la dimension de l'espace

Les contraintes sont alors rassemblée dans une application :

$$C : q \mapsto \begin{pmatrix} C_1 \\ \vdots \\ C_p \end{pmatrix}$$

On remarque alors que :

— La jacobienne J devient une matrice $p \times (n \cdot d)$.

— $JWJ^\intercal\lambda' = -J\dot{q} - \frac{h}{\Delta t}\dot{C}$ est un système d'équations linéaires à p inconnues, que l'on peut résoudre par à l'aide du pivot de Gauss.

— $JWJ^\intercal$ est inversible s'il n'y a pas de redondance entre contraintes ou de contrainte nulle.

— On peut observer la $3^{\text{ème}}$ loi de Newton dans les coefficients de J : une contrainte reliant deux corps entre eux appliquera une force sur les deux corps.

— Les contraintes affectent le déplacement des objets et donc les autres contraintes. Il est naturel de trouver un système linéaire les liant.

3.7 Problèmes rencontrés

Problèmes : Les simulations avec cette méthode ont tendance à perdre de l'énergie lors de fortes inflexions dans l'accélération qui équivalent aux périodes de forte correction du solveur.

Remarque : La correction de Baumgarte n'est pas physiquement réaliste, mais permet de conserver des liens rigides entre les corps.

3.8 Résolution sur l'accélération

Dans le but d'accroître la précision de la simulation, on a souhaité travailler sur l'accélération (ordre 2). Les étapes du programme changent légèrement :

1. Corriger la force appliquée sur l'objet pour satisfaire la contrainte
2. Calculer la position et vitesse à l'aide de l'intégration de Loup Verlet, qui est une méthode d'intégration à l'ordre 4 :

$$q_{n+1} = 2q_n - q_{n-1} + (\Delta t)^2\ddot{q}$$
$$\dot{q}_{n+1} = \frac{q_{n+1} + q_{n-1}}{2\Delta t}$$

3.9 Changement d'ordre

Pour obtenir une contrainte sur l'accélération, on dérive $\dot{C}$

$$\ddot{C} : \dot{J}\dot{q} + J\ddot{q} = 0$$

$\dot{J}\dot{q}$ est une écriture inutilement complexe, pour obtenir ce terme, il est plus facile de passer par l'égalité suivante

$$\boxed{\dot{J}\dot{q} = \frac{d^2C}{dt^2} - J\ddot{q}}$$

3.10 Mise en équation

De la même manière que précédemment, on cherche à isoler λ :

$$\ddot{C} \Leftrightarrow J\ddot{q} + \dot{J}\dot{q} = 0$$
$$\Leftrightarrow J\ddot{q} = -\dot{J}\dot{q}$$
$$\Leftrightarrow JW(F_c + F) = -\dot{J}\dot{q}$$
$$\Leftrightarrow JWF_c = -JWF - \dot{J}\dot{q}$$
$$\Leftrightarrow JWJ^\intercal\lambda = -JWF - \dot{J}\dot{q}$$

3.11 Stabilisation du système

On retrouve une expression identique à l'ordre 1, à l'exception qu'il y a désormais deux termes de stabilisation :

$$\ddot{C}' = \ddot{C} + \frac{h_v}{\Delta t}\dot{C} + \frac{h_p}{\Delta t^2}C$$

Ce qui nous donne l'équation finale

$$\boxed{JWJ^\intercal\lambda = -JWF - \dot{J}\dot{q} - \frac{h_v}{\Delta t}\dot{C} - \frac{h_p}{\Delta t^2}C}$$

3.12 Problèmes rencontrés

Problèmes : Si effectivement cette méthode semble plus précise, elle est beaucoup plus soumise aux erreurs d'arrondis et d'intégration. Les erreurs de positions sont plus importantes, et la stabilisation de Baumgarte n'est pas suffisante pour corriger les erreurs.

Remarque : Des coefficients de stabilisation trop importants peuvent entrainer des oscillations dans le système, voir des divergences. À l'inverse, des coefficients trop faibles provoquent une mauvaise rigidité du système et des pertes d'énergie.

Ce solveur donne de moins bons résultats que le solveur d'ordre 1.

3.13 Mise en place d'un solveur hybride

Le solveur d'ordre 2, bien qu'explosif, est plus endurant que le solveur d'ordre 1 quand toutes les stabilisations sont désactivées. De plus, ils partagent une structure similaire : ils nécessitent tous deux l'inversion de la matrice $JWJ^\intercal$.

Dans le but de diminuer les erreurs de corrections commises par le solveur d'ordre 1 lors de changements brutaux de la vitesse, on a souhaité mettre en place un solveur hybride. Celui-ci chaine les deux solveurs, en utilisant le solveur d'ordre 1 pour les corrections de positions.

3.14 Gain

Ce solveur hybride est plus stable que les solveurs précédents, mais de l'ordre de 1% pour le double du temps de calcul. Il est donc inutile en l'état actuel, car doubler la fréquence d'échantillonnage du solveur d'ordre 1 permet d'obtenir une simulation de meilleure qualité.
On peut alors tenter d'apporter plusieurs améliorations :

- Corriger la force appliquée à l'aide du solveur d'ordre 2
- Intégrer la vitesse à l'aide de la méthode d'Euler implicite
- Corriger la vitesse à l'aide du solveur d'ordre 1
- Intégrer la position à l'aide de la méthode d'Euler implicite.

Problèmes : Utiliser une série de Taylor pour obtenir la position à l'ordre 2 provoque des explosions du système. Or, le solveur calcule déjà la précision nécessaire pour la position donc il n'est pas nécessaire de l'intégrer à l'ordre 2.

Remarque : Quand à la source des erreurs, il s'agit peut-être d'une accumulation avec la stabilisation de Baumgarte a l'ordre 1.

3.15 Série de Taylor

On utilise alors la série de Taylor pour approximer la position et la vitesse à l'ordre 2.

$$q(t+\Delta t) = q(t) + \dot{q}(t)\Delta t + \frac{\ddot{q}(t)}{2}\Delta t^2 + O(\Delta t^3)$$
$$\frac{q(t+\Delta t) - q(t)}{\Delta t} = \dot{q}(t) + \frac{\ddot{q}(t)}{2}\Delta t + O(\Delta t^2)$$
$$\boxed{JWJ^\intercal\lambda' = -J\dot{q} - \frac{1}{2\Delta t}\dot{J}\dot{q}}$$

3.16 Exemple du Pendule simple

Notation : $q = \begin{pmatrix} x \\ y \end{pmatrix}$ est la contrainte usuelle d'un pendule simple en deux dimensions.
$J = \frac{q^\intercal}{||q||}$ sa matrice jacobienne associée.

Exemple :

— Contrainte sur la position : $C : ||\vec{q}|| - l = 0$

— Contrainte sur la vitesse :

$$\frac{dC}{dT} = 0 \Leftrightarrow \frac{d}{dt}(\sqrt{q^2}) = 0$$
$$\Leftrightarrow \frac{2q\cdot\dot{q}}{2\sqrt{q^2}} = 0$$
$$\Leftrightarrow \frac{q^\intercal}{||q||}\dot{q} = 0$$

— Contrainte sur l'accélération :

$$\frac{d^2C}{dt^2} = 0 \Leftrightarrow \frac{d}{dt}\left(\frac{q}{||q||}\cdot\dot{q}\right) = 0$$
$$\Leftrightarrow \frac{d}{dt}\left(\frac{q}{||q||}\right)\cdot\dot{q} + \frac{q}{||q||}\cdot\ddot{q} = 0$$
$$\Leftrightarrow \frac{\dot{q}||q|| - q\left(\frac{q}{||q||}\cdot\dot{q}\right)}{||q||^2}\cdot\dot{q} + \frac{q}{||q||}\cdot\ddot{q} = 0$$
$$\Leftrightarrow \frac{\dot{q}||q||^2 - q(q\cdot\dot{q})}{||q||^3}\cdot\dot{q} + \frac{q}{||q||}\cdot\ddot{q} = 0$$
$$\Leftrightarrow \frac{\dot{q}^2q^2 - (q\cdot\dot{q})^2}{||q||^3} + \frac{q}{||q||}\cdot\ddot{q} = 0$$

— Second terme ordre à l'ordre 2, $\dot{J}\dot{q} = \frac{\dot{q}^2q^2-(q\cdot\dot{q})^2}{||q||^3} = \frac{(x\dot{y}-y\dot{x})^2}{(x+y)^{3/2}}$

3.17 Solveur final actuel

TODO :

4 Simulation fluide

4.1 Choix de la méthode de simulation

Il existe deux méthodes principales pour simuler un fluide : la méthode **Eulérienne** et la méthode **Lagrangienne**.

4.1.1 Description Eulérienne :

On observe le fluide à travers des points fixes dans l'espace (une grille/tableau de cellules). On s'intéresse à la variation des grandeurs physiques (comme la vitesse, la pression, etc.) en fonction du temps et par conséquent à des positions fixes.

- **Avantages** : Permet de facilement d'analyser les champs de vitesse et de pression dans un domaine spatial fixe.
- **Inconvénients** : Peut être moins intuitive pour suivre le mouvement individuel des particules fluides et nécessite des approximations de positions et d'intéractions entre les particules.

4.1.2 Description Lagrangienne :

On suit le mouvement des particules fluides individuelles. On s'intéresse alors à la trajectoire et à l'évolution des propriétés de chaque particule.

- **Avantages** : Permet de suivre précisément le mouvement et l'évolution des particules individuelles.
- **Inconvénients** : Peut être plus complexe à mettre en uvre pour des systèmes de grande échelle ou des écoulements complexes et nécessite de gérer un grand nombre de particules, ce qui peut être très coûteux en termes de calcul.

Ici on choisira d'utiliser la méthode de description **Eulérienne**, car elle est plus adaptée pour les simulations de fluide en grille et permet une visualisation claire des champs de vitesse et de pression.

TODO : Mettre un truc ici

4.2 La grille décalée (Staggered Grid)

Pour simuler efficacement un fluide incompressible en utilisant la description Eulérienne, nous utilisons une structure de données appelée **grille décalée** (ou **staggered grid**).

4.2.1 Principe

Au lieu de stocker toutes les variables physiques (vitesse, pression, densité) aux mêmes points de la grille, on les stocke à des positions décalées :

— **Pression (p) et Densité (d)** : stockée au centre des cellules
TODO : réduire la taille des trucs en gras

— **Composantes de vitesse** : stockées sur les faces des cellules

— Composante u (horizontale) : stockée sur les faces verticales (Est/Ouest)

— Composante v (verticale) : stockée sur les faces horizontales (Nord/Sud)

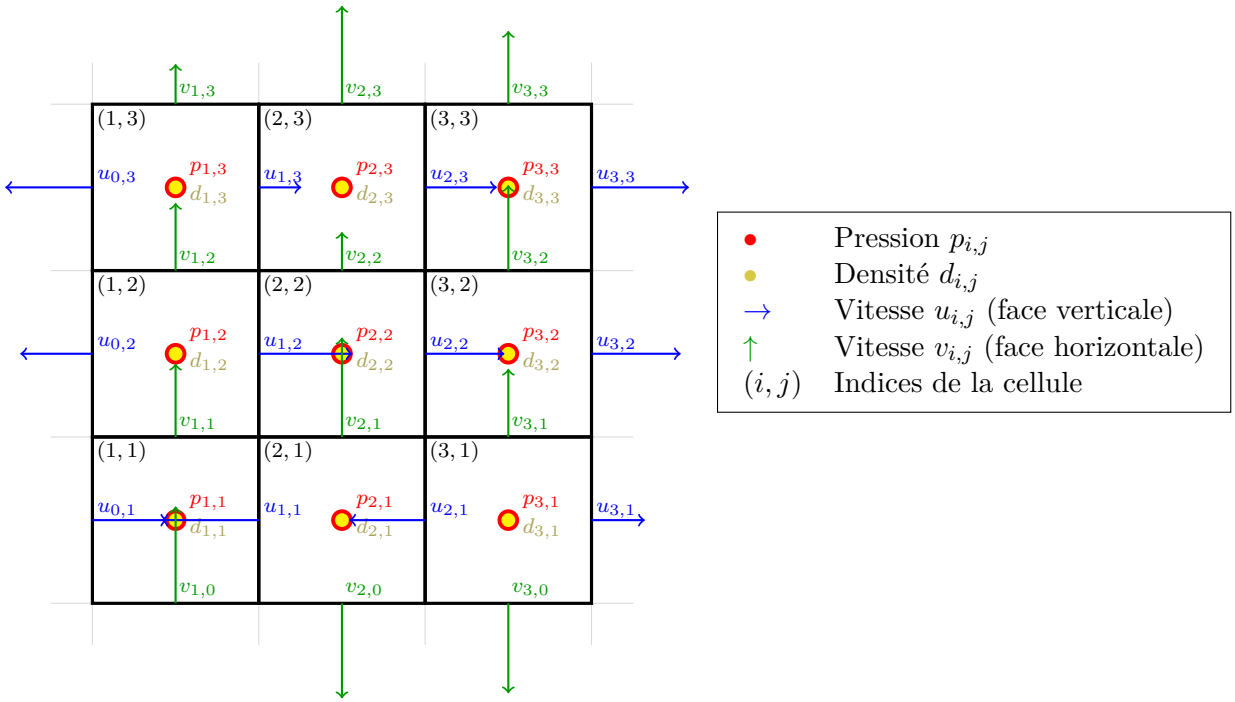

FIGURE 1 – Représentation d'une grille décalée (staggered grid) 2D.

4.3 oui

TODO : trouver le titre aussi

TODO : Décrire l'ordre de fonctionnement du code et les grands principes

4.4 Fonctionnement de objetcs.rs

Ce code fait partie d'un système de simulation de fluide en grille. Il permet de placer aléatoirement des objets solides (`walls`) dans la grille sous différentes formes : cercle, carré, triangle, rectangle.

4.4.1 Fonction principale : place_random_objects

But : Place un nombre donné (`num_objects`) d'objets aléatoires dans la grille.

- Le type d'objet est tiré aléatoirement parmi 4 formes.
- Les positions sont choisies de manière à éviter les bords (via `border_margin`).
- Le type d'objet est tiré aléatoirement parmi : `circle`, `square`, `triangle`, `rectangle`
- Chaque objet est créé via la méthode correspondante, qui remplit certaines cellules avec l'état `wall` = `true`.

4.4.2 Formes

Square, Rectangle Tracent un bloc rectangulaire autour d'un centre donné.

Triangle Trace un triangle équilatéral pointant vers le haut.

- La base s'élargit vers le bas.

Triangle_outline Trace uniquement le contour d'un triangle, sans le remplir.

- Utilise `draw_line` pour dessiner les segments reliant les sommets.

4.4.3 bresenham_line

But : Calcule une ligne discrète entre deux points donnés (x_0, y_0) et (x_1, y_1) sur une grille.
Principe :

- Utilise uniquement des entiers, ce qui le rend rapide et précis pour les maillages 2D.
- L'erreur de tracé est ajustée à chaque étape pour décider s'il faut avancer horizontalement, verticalement ou les deux.
- Retourne un `Vec< (usize, usize) >` contenant tous les points intermédiaires.

Détails techniques :

- dx/dy : décalages absolus en x et y.
- sx/sy : directions de progression (± 1).
- err : erreur cumulative qui guide le choix de la direction à chaque étape.
- $e2$: double de l'erreur actuelle, permet une comparaison sans division.

4.5 Fonctionnement de pressure_computation.rs

Ce code définit une série de méthodes associées à la struct Grid. Il permet de calculer les forces que le fluide exerce sur des objets solides (les "walls"), d'identifier les objets solides dans la grille, de regrouper les forces appliquées à chaque objet, et finalement d'afficher ces forces.

4.5.1 compute_wall_forces

But : Calculer les forces que le fluide exerce localement sur les cellules `wall` (objets solides).
Principe :

— Pour chaque cellule `wall`, on regarde ses voisines immédiates.

— Si une voisine n'est pas un mur, alors elle contient du fluide, et on considère qu'elle applique une force sur le mur.

— Trois types de forces sont modélisées :

  — Force de pression : $-p \times$ normal

  — Force de traînée (*drag*) : proportionnelle à la vitesse normale du fluide

  — Force de cisaillement (*shear*) : proportionnelle à la vitesse tangentielle, pondérée par la viscosité

Le résultat est un vecteur `forces`, qui contient la force subie par chaque cellule `wall`.

4.5.2 identify_objects

But : Identifier les objets solides (constitués de cellules `wall` contiguës) et leur attribuer un identifiant unique (`object_id`).
Principe :

— On parcourt la grille. À chaque fois qu'on trouve un mur non encore étiqueté, on lance un DFS (par pile) pour explorer tout l'objet auquel il appartient.

— Chaque objet reçoit un `object_id` unique (1, 2, 3, ...).

— Le résultat est un vecteur `object_ids`, de la même taille que la grille, contenant l'ID de l'objet pour chaque cellule (ou 0 si ce n'est pas un mur).

4.5.3 compute_object_forces

But : Agréger les forces calculées au niveau des cellules `wall` pour chaque objet entier.
Étapes :

1. Appel à `compute_wall_forces` force appliquée à chaque cellule mur.
2. Appel à `identify_objects` à quel objet appartient chaque mur.
3. Initialisation d'un tableau `objects` de struct `ObjectForce` pour stocker :
   — Force totale.
   — Centre de masse (approximé par les indices i, j).
   — Moment de force (*torque*).
4. Pour chaque cellule `wall`, on accumule :
   — Sa contribution au centre de masse.
   — La force qu'elle subit.
5. Ensuite, pour chaque objet, on calcule le moment de force (*torque*) par produit vectoriel entre :
   — Le vecteur centre de masse cellule.
   — La force exercée sur l'objet.

4.5.4 print_object_forces

But : Affiche dans la console les résultats des calculs précédents.
Affiche pour chaque objet détecté :

— Nombre de cellules.

— Centre de masse.

— Force totale (et sa norme).

— Moment de force.

— Direction approximative (ex : ,).

C'est une fonction de diagnostic ou de visualisation, utile pour savoir si des objets "flottent", "tournent", ou sont "poussés".

4.6 Fonctionnement de condition.rs

Ce code définit un ensemble de constantes globales servant à paramétrer tous les aspects de la simulation de fluide, de l'affichage graphique à la dynamique du fluide, en passant par les sources, le maillage et le flux.

4.6.1 Paramètres de la fenêtre

— **ADAPT_TO_WINDOW** : active l'adaptation automatique de la taille de la fenêtre à la taille de la grille.

— **WINDOW_HEIGHT / WINDOW_WIDTH** : dimensions de la fenêtre, ajustées à la grille si `ADAPT_TO_WINDOW` est vrai, sinon fixées à 720Œ720.

4.6.2 Paramètres de simulation

— **SIM_STEPS** : nombre de pas de temps à simuler.

— **PRINT_FORCES** : affiche ou non les forces calculées pendant la simulation.

— **CIP_CSL4** : active l'advection de densité avec le schéma CIP-CSL4 (non activé ici).

— **DENS_ADV_FAC** : facteur pour l'advection de densité.

— **VEL_STEP** : méthode choisie pour la mise à jour de la vitesse ("1", "2" ou "cip_csl4").

— **PROJECT** : méthode de projection utilisée ("1" ou "2").

— **KARMAN_VORTEX** : tentative d'activer des vortex de Karman.

— **CENTER_SOURCE** : active une source de fluide circulaire au centre de la grille.

— **CENTER_SOURCE_TYPE** : `false` pour une sortie homogène, `true` pour un flux radial type ventilateur.

— **CENTER_SOURCE_RADIUS** : rayon de cette source.

— **CENTER_SOURCE_DENSITY** : densité injectée au centre.

— **CENTER_SOURCE_VELOCITY** : vitesse du fluide injecté par la source centrale.

4.6.3 Paramètres de flux

— **AIR_FLOW** : active un flux d'air global.

— **FLOW_DIRECTION** : direction du flux imposé ("left", "right", "up", "down").

— **FLOW_SPACE** : espacement entre les rangées de flux.

— **FLOW_DENSITY** : densité du fluide dans ce flux.

— **FLOW_VELOCITY** : vitesse de ce flux (0 si `AIR_FLOW` est désactivé).

— **INFLOW_VELOCITY** : forcée égale à `FLOW_VELOCITY` pour garder cohérence.
— **DRAW_VELOCITY_VECTORS** : dessine les vecteurs vitesse à l'écran.
— **VECTOR_SIZE_FACTOR** : facteur d'échelle des flèches de vitesse.
— **PAINT_VORTICITY** : colorie la vorticité (rotation locale du fluide).

4.6.4 Paramètres de la grille

— **EXT_BORDER** : utilise des bordures étendues (à garder `true` si activé).
— **N** : taille de la grille (**Attention : doit être (une puissance de 2) − 2** pour la compatibilité avec l'encodage Morton).
— **DX, DY** : taille (en pixels) d'une cellule en x et y.
— **SIZE** : nombre total de cellules (équivalent à $(N+2)^2$).

4.6.5 Paramètres physiques

— **DT** : pas de temps (en secondes).

4.6.6 Paramètres du fluide

— **VISCOSITY** : viscosité du fluide. Une petite valeur favorise les tourbillons de type Karman.

5 Bibliographie

5.1 Mécanique des fluides

Cours de mécanique des fluides de l'Université Lyon 1 :

Cours de mécanique des fluides - Université Lyon 1

Thèse de BENSEDIRA Sidali de l'université de Blida :

Thèse - Université de Blida

Thèse de COUDERC Frédéric de l'université de Toulouse :

Thèse - Université de Toulouse

TODO :

5.2 Simulation physique

TODO : Donner un nom correcte, Etienne c'est a toi

Thèse de BENSEDIRA Sidali de l'université de Blida :

Baraff_Notes : Physically Based Modeling : Constraints, Chapter 6, 7, 8

Thèse de BENSEDIRA Sidali de l'université de Blida :

Chappuis_Rigid3D : Constraints Derivation for Rigid Body Simulation in 3D

Thèse de BENSEDIRA Sidali de l'université de Blida :

Catto_Solver2009 : Iterative Dynamics with Temporal Coherence (GDC 2009)

Thèse de BENSEDIRA Sidali de l'université de Blida :

Catto_Constraints2014 : Understanding Constraints (GDC 2014)

Thèse de BENSEDIRA Sidali de l'université de Blida :

Catto_Solver2D : A 2D Constraint Solver (Box2D, 2024)

Thèse de BENSEDIRA Sidali de l'université de Blida :

Wiki_GradientConjugué : Méthode du gradient conjugué

Thèse de BENSEDIRA Sidali de l'université de Blida :

Wiki_GaussSeidel : Méthode de Gauss-Seidel