

# Travaux dirigés de Simulation n°2

## Cours de simulation et animation

—IMAC troisième année—

---

### Résolution de la condition d'incompressibilité

#### Objectifs

- Ajout de matériaux solides dans le fluide,
- Construction et résolution du système pour la condition d'incompressibilité.

#### Programme

- Typage des matériaux des cellules,
  - Présentation et construction et construction du système,
  - Résolution.
- 

#### ► Exercice 1. Gérer des types par cellules

##### A - Caractériser les cellules

On appelle type la nature du matériau contenu dans une cellule : entre **fluide**, **vide** (air si simulation de liquide) ou **solide**. Pour l'instant, le booléen `solidWalls` stipule uniquement que l'extérieur de la grille peut être de type **solide** (si `true`) ou **vide** (si `false`).

On souhaite à présent caractériser non seulement les types des bords externes mais aussi des cellules de la grille. On adoptera comme codes pour les types :

- 0 pour une cellule fluide,
- 1 pour une cellule vide,
- 2 pour une cellule solide.

- a) Construisez un tableau `types` de `nbSamples * 4` cases, membre de la classe `Simulation`. Il contiendra les types au centre de chaque cellule de la grille.

```
// in Simulation.hpp
GLuint * types; // Types (0 : cell is fluid,
                // 1 : cell is empty,
                // 2 : cell is solid)

void buildTypes(Object * object);
void drawTypes();

// in Simulation::Simulation(...)
// Samples types
this->types=new GLuint[this->nbSamples];
```

- b) Réalisez l'initialisation dans `void Simulation::initSamples()`. Vous pouvez dans un premier temps, rester à une configuration tout fluide.

```
// in Simulation::initSamples()

types[index]=0;
```

- c) Construisez les fonctions `void Simulation::buildTypes()` et `void Simulation::drawTypes()` appropriées pour le dessin .

```
// Builds an object to visualize the types
// 3D working
void Simulation::buildTypes(Object * object)
{
    std::cout<<"Building on GPU: samples types"<<std::endl;

    object->nbVertices=nbSamples;
    // No indices are necessary since we draw GL_POINTS
    GLuint * indices=NULL;

    GLfloat colors[object->nbVertices*4];
    for (GLuint iSamples=0 ; iSamples<nbSamples ; iSamples++)
    {
        colors[iSamples*4+0]=0.0;
        colors[iSamples*4+1]=0.0;
        colors[iSamples*4+2]=0.0;
        colors[iSamples*4+3]=1.0;
        colors[iSamples*4+types[iSamples]]=1.0;
    }

    // Sends the data into buffers on the GPU
    object->sendPrimitives(samples, indices);
    object->sendColors(colors);
}

// Creates, builds and add to draw list an object for types visualization
void Simulation::drawTypes()
{
    Object * objectTypes=new Object(GL_POINTS);
    GLuint storedObjectTypes=scene->storeObject(objectTypes);
    this->buildTypes(objectTypes);
    GLuint typesID=scene->addObjectToDraw(storedObjectTypes);
    scene->setDrawnObjectShaderID(typesID, defaultShaderID);
}
```

- d) Réalisez la fonction `GLuint Simulation::type(int iX, int iY, int iZ)` qui renvoie le code du type de la cellule dont les coordonnées sont passées en paramètres. cette fonction doit gérer également l'interrogation sur les cellules des `solidWalls`, à l'extérieur de la grille.

```
// Returns type flag for neighbours with provided coordinates (works for boundaries as well)
// 3D not implemented
GLuint Simulation::type(int iX, int iY, int iZ)
{
    if ( (iX<0) || (iX>=(int)nbSamplesX) || (iY<0) || (iY>=(int)nbSamplesY) )
    {
        if (solidWalls) return 2;
        else return 1;
    }
    return types[iY*nbSamplesX+iX];
}
```

## B - Renforcer les conditions aux bords

Maintenir la simulation en mode `solidwalls` a consisté pour l'instant à annuler systématiquement les composantes normales aux bords de la vitesse.

Un tel effort est nécessaire à l'initialisation et à chaque mise à jour des vitesses, soit dans :

- `initVelocitiesBorders()`,
- `advectVelocities(...)`,
- `applyForces(...)`,
- `project(...)`.

Puisque les types permettent maintenant de multiplier les cellules solides, renforcer les conditions aux bords devient plus complexe : la séquence des actions à réaliser peut être rassembler dans une fonction.

- a) Réalisez la fonction `void enforceVelocitiesBorders()`; qui renforce les conditions aux bords de tous les solides.

```

// Cancels normal velocities on solid boundaries
// 3D not implemented
void Simulation::enforceVelocitiesBorders()
{
    for (GLuint iY=0 ; iY<nbSamplesY ; iY++)
    {
        for (GLuint iX=0 ; iX<nbSamplesX ; iX++)
        {
            GLuint iSamples=iY*nbSamplesX+iX;
            GLuint indexVelocitiesXLeft =iY *(nbSamplesX+1)+ iX;
            GLuint indexVelocitiesXRight =iY *(nbSamplesX+1)+(iX+1);
            GLuint indexVelocitiesYBottom=iY * nbSamplesX + iX;
            GLuint indexVelocitiesYTop =(iY+1) * nbSamplesX + iX;

            // If solid cell
            if (types[iSamples]==2)
            {
                // normal velocity component must be null
                velocitiesX[indexVelocitiesXLeft] =0.0;
                velocitiesX[indexVelocitiesXRight] =0.0;
                velocitiesY[indexVelocitiesYBottom]=0.0;
                velocitiesY[indexVelocitiesYTop] =0.0;
            }

            if (solidWalls)
            {
                if (iX==0) velocitiesX[indexVelocitiesXLeft] =0.0;
                if (iX==(nbSamplesX-1)) velocitiesX[indexVelocitiesXRight] =0.0;
                if (iY==0) velocitiesY[indexVelocitiesYBottom]=0.0;
                if (iY==(nbSamplesY-1)) velocitiesY[indexVelocitiesYTop] =0.0;
            }
        }
    }
}

```

b) Appelez là à la place des lignes actuelles réalisant la tâche sur les solidWalls.

```

// in Simulation::initVelocitiesBorders()
// If boundaries are solid, normal velocity component must be null
//if ( ( solidWalls) && ((iX==0) || (iX==nbSamplesX)) )
// velocitiesX[iBordersX]=0.0;

// If boundaries are solid, normal velocity component must be null
//if ( ( solidWalls) && ((iY==0) || (iY==nbSamplesY)) )
// velocitiesY[iBordersY]=0.0;
enforceVelocitiesBorders();

// in Simulation::advectVelocities(GLfloat dt)
// If boundaries are solid, normal velocity components must be null
/*if (solidWalls)
{
    for (GLuint iY=0 ; iY<nbSamplesY ; iY++)
        for (GLuint iX=0 ; iX<(nbSamplesX+1) ; iX++)
            if ((iX==0) || (iX==nbSamplesX))
                velocitiesX[iY*(nbSamplesX+1)+iX]=0.0;

    for (GLuint iY=0 ; iY<(nbSamplesY+1) ; iY++)
        for (GLuint iX=0 ; iX<nbSamplesX ; iX++)
            if ((iY==0) || (iY==nbSamplesY))
                velocitiesY[iY*nbSamplesX+iX]=0.0;
}*/
enforceVelocitiesBorders();

// in Simulation::applyForces(GLfloat dt) : end
enforceVelocitiesBorders();

// in Simulation::project(GLfloat dt)
enforceVelocitiesBorders();

```

c) Dans le même esprit, il serait sûrement raisonnable, pour raccourcir le code, de capitaliser en fonctions les mises à jours des différents VBO. De plus, cette fois pour optimiser le framerate, il serait intéressant de ne réaliser les constructions et mises à jour de ces VBOs que lorsque l'on est en mode debug (booléen à créer dans `Simulation`), c'est à dire quand on a besoin de l'affichage de ces données pour faire des tests. Ne réalisez pas ces tâches pendant la séance, pour ne pas perdre de temps.

## ► Exercice 2. Poser le problème

### A - Pression et incompressibilité

Dans la suite,  $vX$  et  $vY$  remplacent *velocitiesX* et *velocitiesY*. *Border* est remplacé par *Br*.

Le terme  $-\frac{1}{\rho}\nabla p$  indique la manière dont le gradient de pression agit sur la vitesse de sorte que les zones de hautes pressions poussent vers les zones de basses pression.

$$\begin{aligned} vX_{iX-\frac{1}{2}} &= vX_{iX-\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iX} - p_{iX-1}}{\Delta x} & vX_{iX+\frac{1}{2}} &= vX_{iX+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iX+1} - p_{iX}}{\Delta x} \\ vY_{iY-\frac{1}{2}} &= vY_{iY-\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iY} - p_{iY-1}}{\Delta x} & vY_{iY+\frac{1}{2}} &= vY_{iY+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iY+1} - p_{iY}}{\Delta x} \end{aligned}$$

`project()` est l'étape du calcul qui évalue la pression en chaque cellule pour modifier en conséquence la vitesse. Vous avez déjà programmé ce calcul mais vous êtes heurtés à un problème, la pression est une inconnue.

On souhaite rendre le fluide incompressible. Mathématiquement, cela revient à dire qu'on veut que le champ de vitesse finale soit non divergent :

$$\frac{vX_{iX+\frac{1}{2}} - vX_{iX-\frac{1}{2}}}{\Delta x} + \frac{vY_{iY+\frac{1}{2}} - vY_{iY-\frac{1}{2}}}{\Delta x} = 0$$

La pression dépend des conditions de bords, des vitesses sur la grille et de la condition d'incompressibilité.

### B - Résolution locale (par cellule)

Pour formuler la pression selon les quantités connues que sont les vitesses courantes et la divergence courante, mélangeons les deux ensembles de formules (cités plus haut).

$$\frac{(vX_{iX+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iX+1} - p_{iX}}{\Delta x}) - (vX_{iX-\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iX} - p_{iX-1}}{\Delta x})}{\Delta x} + \frac{(vY_{iY+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iY+1} - p_{iY}}{\Delta x}) - (vY_{iY-\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{iY} - p_{iY-1}}{\Delta x})}{\Delta x} = 0$$

$$\frac{\Delta t}{\rho} \left( \frac{4p(iX, iY) - p(iX+1, iY) - p(iX-1, iY) - p(iX, iY+1) - p(iX, iY-1)}{\Delta x^2} \right) = - \left( \frac{vX_{iX+\frac{1}{2}} - vX_{iX-\frac{1}{2}}}{\Delta x} + \frac{vY_{iY+\frac{1}{2}} - vY_{iY-\frac{1}{2}}}{\Delta x} \right)$$

$$\frac{\Delta t}{\rho \Delta x^2} (4p_{cell} - p_{right} - p_{left} - p_{top} - p_{bottom}) = - \frac{1}{\Delta x} (vX_{rightBr} - vX_{leftBr} + vY_{topBr} - vY_{bottomBr})$$

Il s'agit de l'équation à résoudre pour une cellule.

### C - Résolution globale

Comme les cellules sont interdépendantes, il est impossible de toutes les résoudre séquentiellement. On doit donc procéder à la recherche d'une solution optimale sur l'ensemble de la grille. Cela revient à résoudre un système fait de cette équation appliquée à chaque cellule :

$$Ap = -(\nabla \cdot \vec{u})$$

avec  $p$  le vecteur de taille (*nbSamples*) des pressions par cellules et  $A$  une matrice de taille (*nbSamples* \* *nbSamples*). Chaque ligne de  $A$  correspond à une cellule et à l'équation à résoudre pour trouver la pression dans cette cellule.

Plus précisément l'intersection d'une ligne  $a$  avec une colonne  $b$  correspond au coefficient de la pression s'exerçant entre les deux cellules correspondantes. Les interactions n'existant qu'entre cellules directement voisines, la matrice est essentiellement remplie de 0 (matrice "**sparse**").

La diagonale (intersection d'une cellule  $a$  avec elle-même) est remplie des coefficients  $n \frac{\Delta t}{\rho \Delta x^2}$  qui pondère la pression de cette cellule  $a$ . Les 4 (ou 6 en 3D) autres valeurs non nulles par lignes contiennent les coefficients  $-\frac{\Delta t}{\rho \Delta x^2}$  des pressions s'exerçant sur les cellules directement voisines de  $a$ .

## D - Conditions de bord

Si des cellules voisines de  $a$  ne sont pas de type fluide,  $n$  diminue d'autant et les coefficients de pressions correspondants sur la ligne sont à modifier.

- Si une cellule est de type vide, quelle pression est exercée sur une cellule voisine liquide ?  
Une pression nulle. Cela signifie juste que la cellule liquide aura un coefficient nul supplémentaire sur sa ligne dans  $A$  (et  $n < 4$ ).
- Si une cellule est de type solide, quelle pression est exercée sur une cellule voisine liquide ?  
Une pression telle que la vitesse au bord est celle connue pour le solide (condition de bord). Par exemple, pour un mur vertical avec un solide, on peut obtenir la pression "virtuelle" pour le solide (en réalité à la frontière avec celui-ci) en utilisant  $vX_{solid}$ .

$$vX_{solid} = vX_{br} - \Delta t \frac{1}{\rho} \frac{p_{solid} - p_{fluide}}{\Delta x}$$

$$p_{solid} = p_{fluide} + \frac{\rho \Delta x}{\Delta t} (vX_{br} - vX_{solid})$$

Si le solide est inanimé (comme les solidWalls que nous avons utilisé jusqu'à présent) :  $vX_{solid} = 0$

Dans ce second cas, on doit remplacer  $p_{(iX,iY)}$  de cette cellule solide dans les 4 (ou 6) équations de ces voisins. Par exemple si le solide est à droite de la cellule  $a$ , son équation devient :

$$\frac{\Delta t}{\rho \Delta x^2} (4p_{cell} - [p_{cell} + \frac{\rho \Delta x}{\Delta t} (vX_{rightBr} - vX_{solid})] - p_{left} - p_{top} - p_{bottom}) = -(\nabla \cdot \vec{u})$$

Q'on peut également écrire :

$$\frac{\Delta t}{\rho \Delta x^2} (np_{cell} - p_{left} - p_{top} - p_{bottom}) = -(\nabla \cdot \vec{u}) + (\frac{vX_{rightBr} - vX_{solid}}{\Delta x})$$

où  $n$  est le nombre de cellules voisines de  $a$  et de type fluide ou air (non solide), ici 3.

Si le solide avait été à gauche, la formule aurait été :

$$\frac{\Delta t}{\rho \Delta x^2} (np_{cell} - p_{right} - p_{top} - p_{bottom}) = -(\nabla \cdot \vec{u}) - (\frac{vX_{leftBr} - vX_{solid}}{\Delta x})$$

(Noter le signe qui diffère.)

## E - Exemple

Voici un exemple du système à résoudre pour une grille de 9 cellules, entourée de bords solides, dont la cellule en haut au centre est vide et dont la cellule en haut à droite est solide.

		solid	solid	solid
solid	fluid	empty	solid	solid
solid	fluid	fluid	fluid	solid
solid	fluid	fluid	fluid	solid
		solid	solid	solid

$$\begin{bmatrix}
2k & -k & 0 & -k & 0 & 0 & 0 & 0 & 0 \\
-k & 3k & -k & 0 & -k & 0 & 0 & 0 & 0 \\
0 & -k & 2k & -k & 0 & -k & 0 & 0 & 0 \\
-k & 0 & -k & 3k & -k & 0 & -k & 0 & 0 \\
0 & -k & 0 & -k & 4k & -k & 0 & \textcolor{red}{0} & 0 \\
0 & 0 & -k & 0 & -k & 2k & -k & 0 & \textcolor{red}{0} \\
0 & 0 & 0 & -k & 0 & -k & 2k & \textcolor{red}{0} & 0 \\
0 & 0 & 0 & 0 & \textcolor{red}{0} & 0 & \textcolor{red}{0} & 2k & \textcolor{red}{0} \\
0 & 0 & 0 & 0 & 0 & \textcolor{red}{0} & 0 & \textcolor{red}{0} & 2k
\end{bmatrix}
\begin{bmatrix}
p_{(0,0)} \\
p_{(1,0)} \\
p_{(2,0)} \\
p_{(0,1)} \\
p_{(1,1)} \\
p_{(2,1)} \\
p_{(0,2)} \\
p_{(1,2)} \\
p_{(2,2)}
\end{bmatrix}
=
\begin{bmatrix}
-div(0,0) \\
-div(1,0) \\
-div(2,0) \\
-div(0,1) \\
-div(1,1) \\
-div(2,1) + (\frac{vY_{topBr}}{\Delta x}) \\
-div(0,2) \\
-div(1,2) + (\frac{vX_{rightBr}}{\Delta x}) \\
-div(2,2)
\end{bmatrix}$$

avec  $k = \frac{\Delta t}{\rho \Delta x^2}$

En plus du fait que la matrice est essentiellement vide, on remarque la symétrie autour de la diagonale ( $p(a,b) = p(b,a)$ ). Chaque cellule (ou ligne) peut en fait être résumée à valeurs : la diagonale, lien avec la droite

et le lien avec le haut. Il suffit donc de 3 vecteurs `Adiag[nbsamples]`, `Aright[nbsamples]` et `Atop[nbsamples]` pour stocker toute la matrice.

Voici leurs valeurs pour l'exemple :

$$\begin{aligned} \text{Adiag} &= \begin{bmatrix} 2k & 3k & 2k & 3k & 4k & 2k & 2k & 2k & 2k \end{bmatrix} \\ \text{Aright} &= \begin{bmatrix} -k & -k & -k & -k & -k & -k & \mathbf{0} & -k & \mathbf{0} \end{bmatrix} \\ \text{Atop} &= \begin{bmatrix} -k & -k & -k & -k & \mathbf{0} & \mathbf{0} & 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

Que ce soit dans  $A$ ,  $p$  ou à droite du système, il est en fait inutile de remplir les lignes correspondant aux cellules vides ou solides.

## F - Programmation du système

- Dans `project()`, allouez et initialisez à 0 les vecteurs `rhs` (Right Hand Side : côté droit du système), `Adiag`, `Aright` et `Atop`. Pour éviter trop d'erreurs d'arrondi lors des calculs sur ces vecteurs, utilisez des `double`.
- Remplissez ces 4 vecteurs.
  - Initialisez `rhs` avec votre fonction `setDivergences(...)`.
  - Pour chaque cellule :
    - Stockez le type de chacun des quatre voisins potentiels,
    - Inversez le signe de `rhs`,
    - Pour chaque voisin solide, modifiez `rhs` en conséquence,
    - Pour chaque cellule fluide, incrémentez `Adiag` de  $k = \frac{\Delta t}{\rho \Delta x^2}$  à chaque voisin non solide.
    - Pour chaque voisin haut/droit fluide, remplissez `Aright/Atop` par  $-k$ .

## ► Exercice 3. Résoudre le système

### A - Choix d'une méthode

Les propriétés d'une matrice au sein d'un système que l'on souhaite résoudre conditionnent le choix de la meilleure méthode de résolution (selon ses propriétés d'occupation mémoire, sa rapidité de convergence lorsqu'il s'agit d'une méthode itérative, l'exactitude de ses résultats ou le compromis temps/qualité, sa capacité à monter en échelle...). Dans notre cas, en plus de ces paramètres, entre en jeu la simplicité de programmation et la robustesse à des configurations aux limites variées et irrégulières.

### B - Méthodes itératives

la matrice  $A$  est susceptible d'atteindre de très grandes tailles et est très vide (ou `sparse`). Les systèmes employant de telles matrices sont souvent résolus par des algorithmes itératifs. Cela signifie que la solution est approchée à chaque itération de l'algorithme, que l'on stoppe lorsque l'erreur passe sous un certain seuil et devient acceptable selon le standard qu'on s'est fixé. En synthèse d'image, cette méthodologie prend tout son sens puisqu'elle permet de calibrer le ratio temps/qualité de manière très souple selon l'application, juste en modifiant la condition d'arrêt. Une méthode itérative est particulièrement adaptée aux grandes matrices `sparse` parcequ'elle a l'avantage de ne nécessiter que peu de stockage d'information et donc de facilement monter en échelle, à l'inverse d'une méthode directe qui remplirait assez vite la mémoire en cas de grosses matrices.

### C - Conjugate gradient

La matrice  $A$  est, de plus, de type symétrique, positive, (semi-)définie. En tant que telle, nous avons accès à la méthode itérative du **gradient conjugué** ou **conjugate gradient**. Cette méthode a l'avantage d'être très simple à implémenter, peu coûteuse en nombre et complexité d'opérations par itération, élégante et "relativement" intuitive... Elle est aussi robuste aux domaines irréguliers. Une explication intuitive du conjugate gradient vous sera présentée au prochain TD, si vous êtes intéressés.

### D - Preconditionned Conjugate Gradient

Malheureusement, la méthode prend d'autant plus d'itérations pour converger vers la solution que la grille est large. On emploie donc souvent une amélioration qui consiste à "préconditionner le système", c'est à dire à deviner, à partir de  $A$  une solution de départ la plus proche possible de la solution, pour réduire le nombre des itérations. Le preconditionneur que nous utilisons ici s'intitule "**Modified Incomplete Choleski (Order 0)**". Concrètement, utiliser un preconditionneur consiste à multiplier les deux côtés de l'équation  $A$  par une matrice  $M$  qu'on estime être proche de  $A^{-1}$ . Si  $M$  est bien proche de  $A^{-1}$ , alors  $MA$  sera proche de l'identité, et la résolution convergera très vite. Finalement, multiplier à chaque itération le vecteur des nouvelles pressions  $p$  et  $rhs$  par  $M$  permet bien d'atteindre plus rapidement la solution.

## E - Condition d'arrêt

Si nous voulions évaluer l'erreur à chaque pas et donc pouvoir nous arrêter lorsqu'elle passe sous un certain seuil, il faudrait mesurer l'écart entre la solution à l'étape  $i$  et la solution exacte. Il faudrait donc avoir la solution exacte...

On utilise donc une autre mesure, très satisfaisante, qui est le résidu  $r_i = b - Ap_i$ . On voit bien que le résidu constitue une sorte de reste une fois la pression  $i$  appliquée dans le système. Il tend vers 0 lors  $p$  tend vers la solution et  $i$  vers l'infini.

Encore plus simplement, il se trouve que le résidu est en fait la divergence une fois les pressions appliquées. Puisque l'on cherche un champ de vitesse non-divergent, il suffit de savoir quelle divergence finale on est près à tolérer pour savoir quand arrêter l'algorithme. Ce seuil de tolérance doit être choisi petit pour préserver l'exactitude de la solution mais aussi suffisamment grand pour que chaque résolution (il y en a une par frame) ne dure pas trop longtemps. Une valeur de très bonne qualité serait de l'ordre de  $tol = 10^{-6}$ .

En plus de cette condition d'arrêt, il est nécessaire de borner le nombre d'itérations. En effet, lorsque les erreurs sur les flottants s'accumulent, l'algorithme converge moins bien. On arrête donc manuellement les itérations : après par exemple, la centième. Ainsi, on évite tout risque de boucle infinie qui s'éloignerait petit à petit de la solution.

## F - Programmation

A chaque itération de l'algorithme du conjugate gradient ont lieu :

- une multiplication de matrice sparse avec un vecteur,
- des multiplications et additions de scalaires à des vecteurs,
- quelques produits scalaires.

Les vecteurs pouvant atteindre de grandes tailles, il est bénéfique d'employer une bibliothèque spécialisée dans ce type de calcul. Les bibliothèques de type "BLAS" sont optimisées pour exploiter au mieux des instructions bas niveau optimale pour ces calculs. Nous avons choisi d'utiliser la bibliothèque "**Eigen**", qui annonce une efficacité importante et une extrême simplicité d'utilisation.

- a) Téléchargez **Eigen 3.0.3** sur <http://bitbucket.org/eigen/eigen/get/3.0.3.tar.bz2>. Décompressez l'archive dans votre dossier `api`.

Ajoutez l'include dans `Application.hpp` :

```
#include <../api/eigen-eigen-3.0.3/Eigen/Dense>
```

La documentation de la librairie est disponible sur [http://eigen.tuxfamily.org/index.php?title=Main\\_Page](http://eigen.tuxfamily.org/index.php?title=Main_Page)

- b) Réalisez la fonction `void Simulation::multiplySparseMatrix(double * Adiag, double * Aright, double * Atop, Eigen::VectorXd v, Eigen::VectorXd * result)` qui multiplie la matrice sparse `A` au vecteur `v` et stocke le résultat dans `result`.
- c) Recopiez cette fonction qui réalise le conjugate gradient.

```
// Fills pressures from sparse matrix and rhs=b
void Simulation::conjugateGradient(double * Adiag, double * Aright, double * Atop, double * b)
{
    GLuint maxIterations=100;
    double tol=1e-4;

    // Initializes pressures to null
    Eigen::VectorXd p=Eigen::VectorXd::Zero(nbSamples);
    // Copy r from b and return if r null
    Eigen::VectorXd r(nbSamples);

    for (GLuint iSamples=0 ; iSamples<nbSamples ; iSamples++)
        r[iSamples]=b[iSamples];

    double rInfinityNorm=r.lpNorm<Eigen::Infinity>();
    if (rInfinityNorm<=tol)
    {
        for (GLuint iSamples=0 ; iSamples<nbSamples ; iSamples++)
            pressures[iSamples]=(GLfloat)p[iSamples];
        //std::cout<<"End reached immediately."<<std::endl;
        //std::cout<<"r="<<rInfinityNorm<<std::endl;
        return;
    }
}
```

```

// Builds Preconditionner
Eigen::VectorXd precon=Eigen::VectorXd::Zero(nbSamples);
MICPreconditioner(Adiag, Aright, Atop, &precon);

// Apply preconditionner from r to z
Eigen::VectorXd z=Eigen::VectorXd::Zero(nbSamples);
applyPreconditioner(Aright, Atop, precon, r, &z);
// Copy z to s
Eigen::VectorXd s(z);
double sigma=z.dot(r);
for (GLuint i=0 ; i<maxIterations ; i++)
{
    // Multiply matrix A to s to get z;
    multiplySparseMatrix(Adiag, Aright, Atop, s, &z);
    double alpha;
    double div=z.dot(s);
    if (div!=0.0) alpha=sigma/div;
    p+=alpha*s;
    r-=alpha*z;
    rInfinityNorm=r.lpNorm<Eigen::Infinity>();
    if (rInfinityNorm<=tol)
    {
        i=maxIterations;
    }
    else
    {
        applyPreconditioner(Aright, Atop, precon, r, &z);
        double beta;
        double newSigma=z.dot(r);
        if (sigma!=0.0) beta=newSigma/sigma;
        s=z+beta*s;
        sigma=newSigma;
    }
}
for (GLuint iSamples=0 ; iSamples<nbSamples ; iSamples++)
    pressures[iSamples]=(GLfloat)p[iSamples];
}

```

d) Recopiez ces fonctions qui calculent et appliquant le préconditionneur.

```

Choleski (level 0) preconditioner
void Simulation::MICPreconditioner(double * Adiag, double * Aright, double * Atop, Eigen::VectorXd
* precon)
{
    double tau=0.97;
    double safetyConstant=0.25;
    for (int iY=0 ; iY<(int)nbSamplesY ; iY++)
    {
        for (int iX=0 ; iX<(int)nbSamplesX ; iX++)
        {
            int iS=iY*nbSamplesX+iX;
            int iSLeft=iY*nbSamplesX+(iX-1);
            int iSBottom=(iY-1)*nbSamplesX+iX;
            double e=Adiag[iS];
            if (iX>0)
            {
                e-=pow(Aright[iSLeft]*(*precon)[iSLeft], 2)
                +tau*(Aright[iSLeft] *Atop[iSLeft] *pow((*precon)[iSLeft] , 2));
            }
            if (iY>0)
            {
                e-=pow(Atop[iSBottom]*(*precon)[iSBottom], 2)
                +tau*(Atop[iSBottom]*Aright[iSLeft]*pow((*precon)[iSBottom], 2));
            }
            if (e<(safetyConstant*Adiag[iS])) e=Adiag[iS];
            if (e!=0.0) (*precon)[iS]=1.0/sqrt(e);
        }
    }
}

// Multiplies preconditionneur to r and stores it in z
void Simulation::applyPreconditioner(double * Aright, double * Atop, Eigen::VectorXd precon,
Eigen::VectorXd r, Eigen::VectorXd * z)
{
    double t=0.0;

```



```

Eigen::VectorXd q=Eigen::VectorXd::Zero(nbSamples);
for (int iY=0 ; iY<(int)nbSamplesY ; iY++)
{
    for (int iX=0 ; iX<(int)nbSamplesX ; iX++)
    {
        int iS=iY*nbSamplesX+iX;
        int iSLeft=iY*nbSamplesX+(iX-1);
        int iSBottom=(iY-1)*nbSamplesX+iX;
        t=r[iS];
        if (iX>0) t-=Aright[iSLeft]*precon[iSLeft] *q[iSLeft];
        if (iY>0) t-=Atop[iSBottom]*precon[iSBottom]*q[iSBottom];
        q[iS]=t*precon[iS];
    }
}
for (int iY=((int)nbSamplesY-1) ; iY>=0; iY--)
{
    for (int iX=((int)nbSamplesX-1) ; iX>=0 ; iX--)
    {
        int iS=iY*nbSamplesX+iX;
        int iSRight=iY*nbSamplesX+(iX+1);
        int iSTop=(iY+1)*nbSamplesX+iX;
        t=q[iS];
        if (iX<((int)nbSamplesX-1)) t-=Aright[iS]*precon[iS]*(*z)[iSRight];
        if (iY<((int)nbSamplesY-1)) t-=Atop[iS] *precon[iS]*(*z)[iSTop];
        (*z)[iS]=t*precon[iS];
    }
}
}

```

- e) Testez votre simulation. Si elle fonctionne, vous devez constater que la condition d'incompressibilité est bien apparente. Essayez plusieurs valeurs pour `tol` et observez la différence en qualité et en nombre d'itérations.