17066

Modélisation de la trajectoire d'une fusée

http://french.peopledaily.com.cn

Sommaire

- Premières modélisations
- Fusée à eau
- Simulation de fluides incompressibles

Premières modélisations

Forces:

- Poids: $-m(t) \cdot g$
- Force de frottement fluide: $-\alpha \cdot v(t)$
- Poussée: $D \cdot u$

$$\frac{d}{dt}\big(m(t) \cdot v(t)\big) = -m(t) \cdot g - \alpha \cdot v(t) + D \cdot u$$

- Résolution de l'équation différentielle:

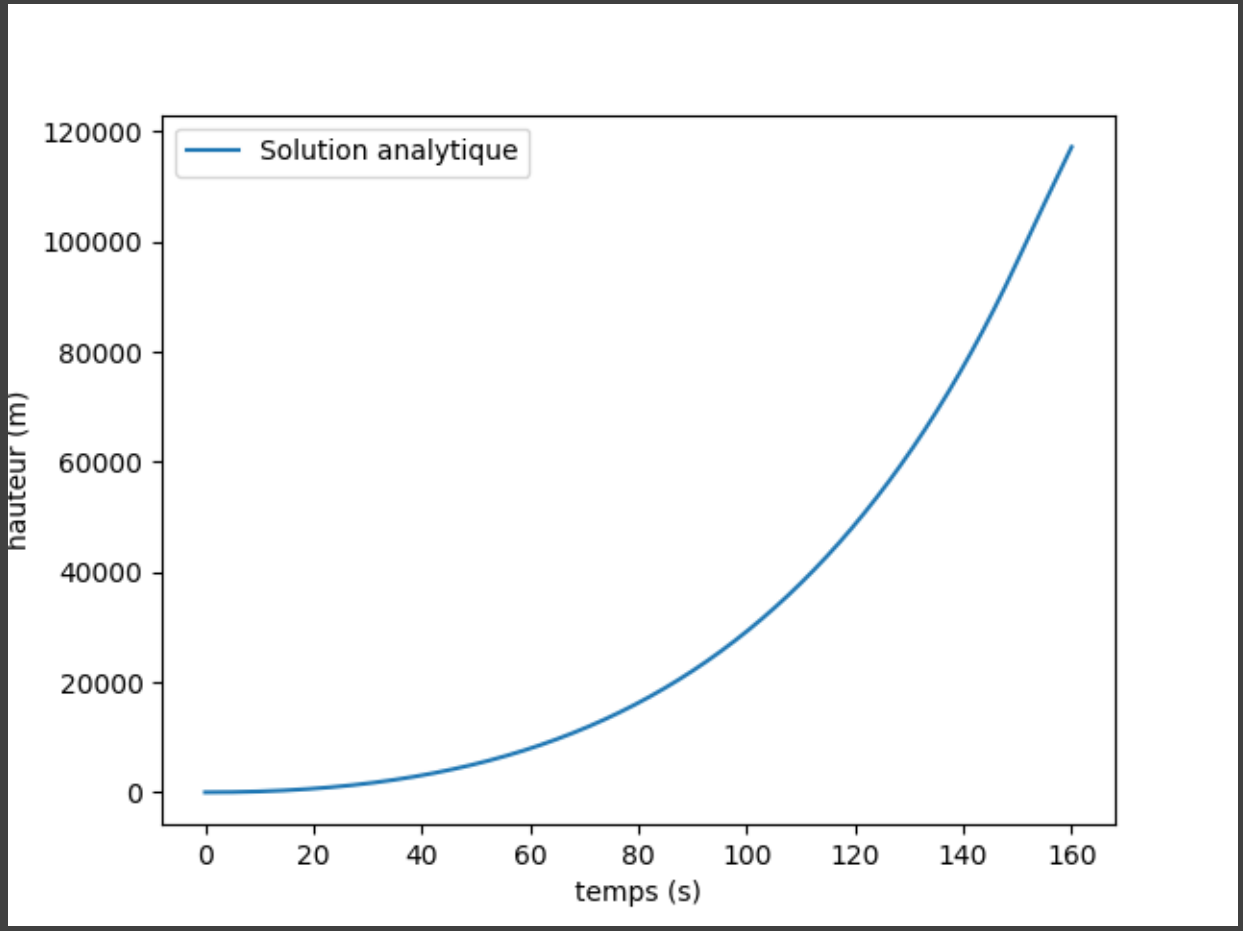
- $v(\mathrm{t}) = -\frac{g\cdot(M_0 - D\cdot t)}{\alpha - D} + \frac{D\cdot u}{\alpha} + A\cdot(M_0 - D\cdot t)^{\frac{\alpha}{D}}$

- $z(t) = \frac{g\cdot(M_0 - D\cdot t)^2}{(\alpha - D)\cdot 2\cdot D} + \frac{D\cdot u\cdot t}{\alpha} - \frac{A\cdot(M_0 - D\cdot t)^{\frac{\alpha}{D}+1}}{\alpha + D} + B$

  - Avec $A = -\left(\frac{g\cdot M_0}{D-\alpha} + \frac{D\cdot u}{\alpha}\right)\cdot\frac{1}{{M_0}^{\frac{\alpha}{D}}}$

    - Et $B = \frac{g\cdot {M_0}^2}{2\cdot D\cdot(D-\alpha)} + \frac{A\cdot {M_0}^{\frac{\alpha}{D}+1}}{\alpha + D}$

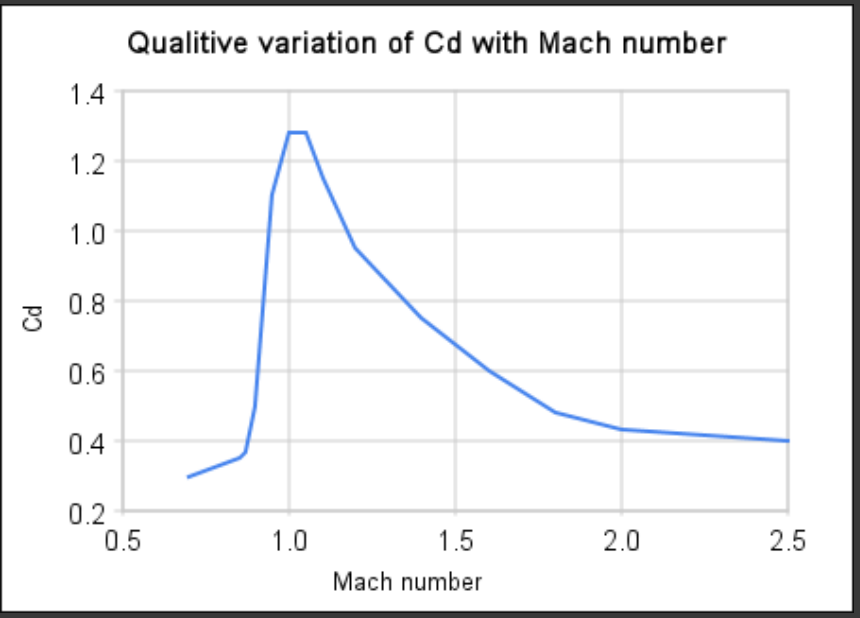
- Données de la fusée Saturn V:
  - Masse à vide : 718 tonnes
  - Masse de carburant : 2279 tonnes
  - Vitesse d’éjection : 2500 m/s

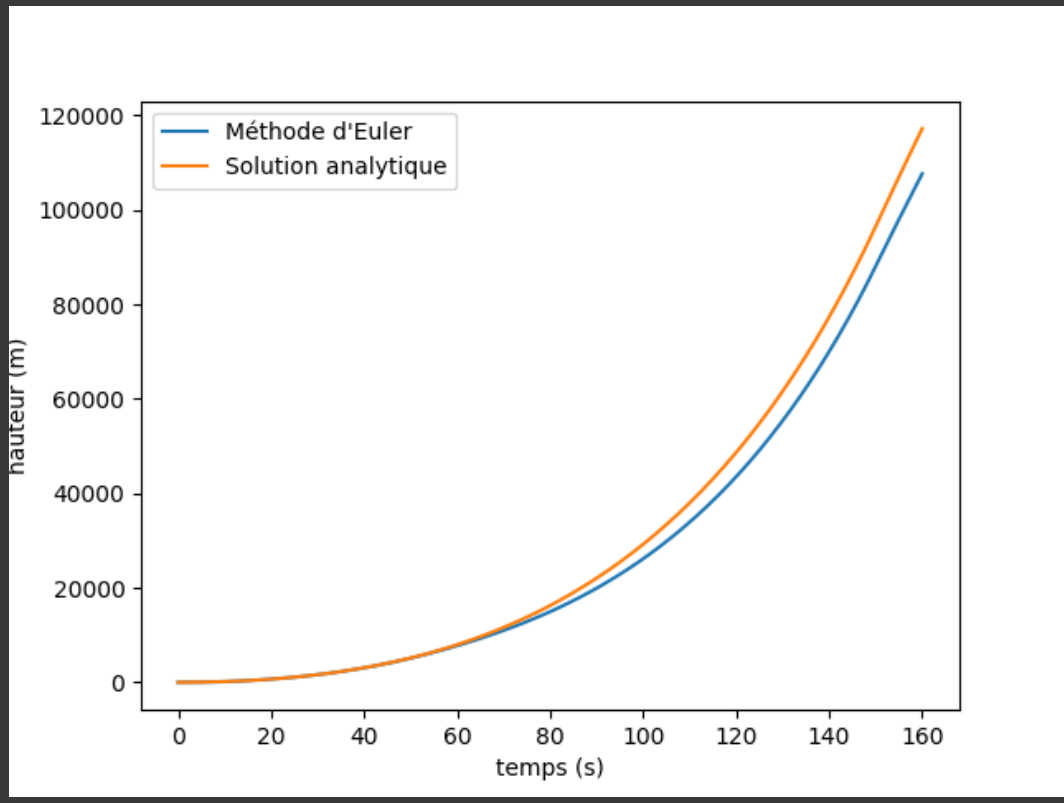


- Utilisation d’une force de frottement fluide de la forme:

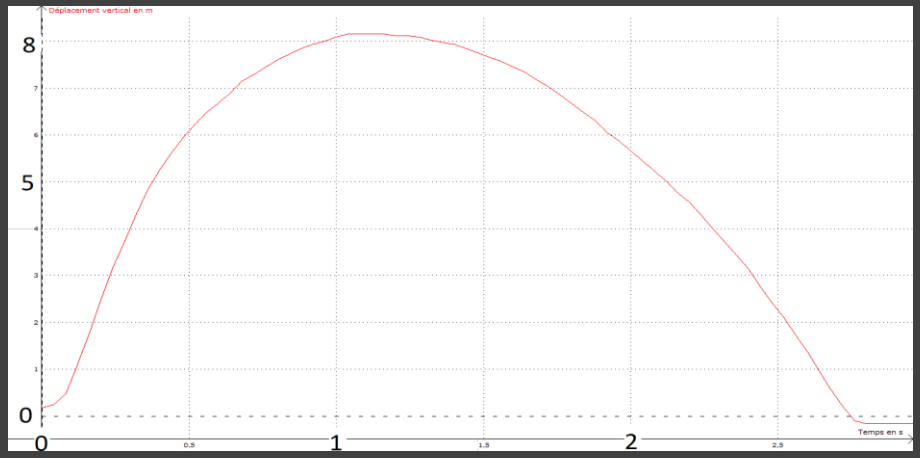
$$\mathrm{F} = -\frac{1}{2}\rho \cdot C_d \cdot S \cdot v^2$$

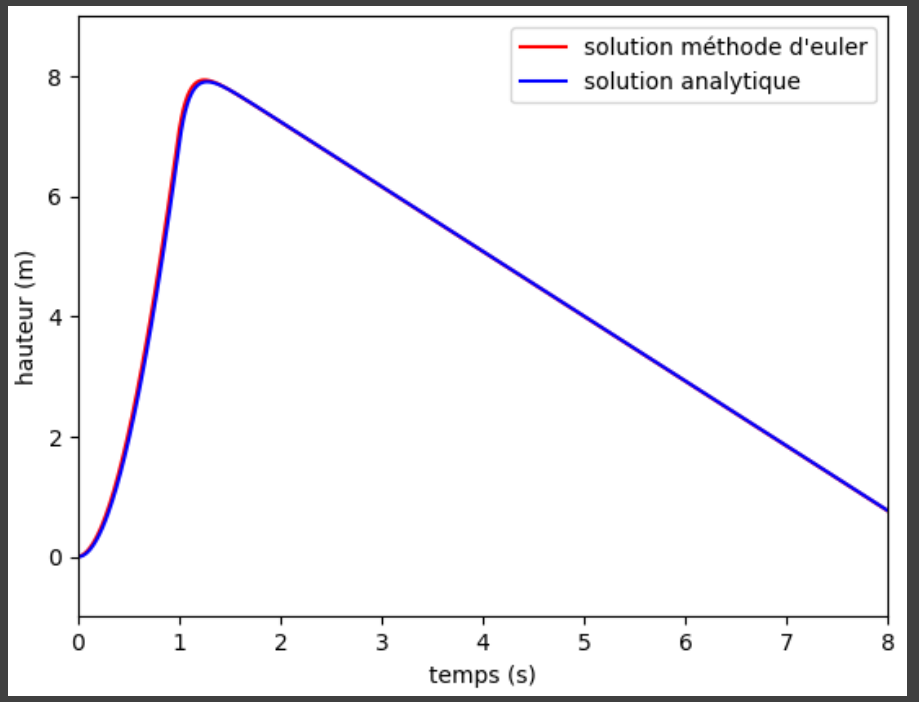
- Évolution du coefficient C_d selon la vitesse:





La fusée à eau





Détermination du coefficient de frottement

- Simulation d’une soufflerie

- Code adapté d’un algorithme par les créateurs de Maya

Simulation de fluide

Pour un fluide incompressible:

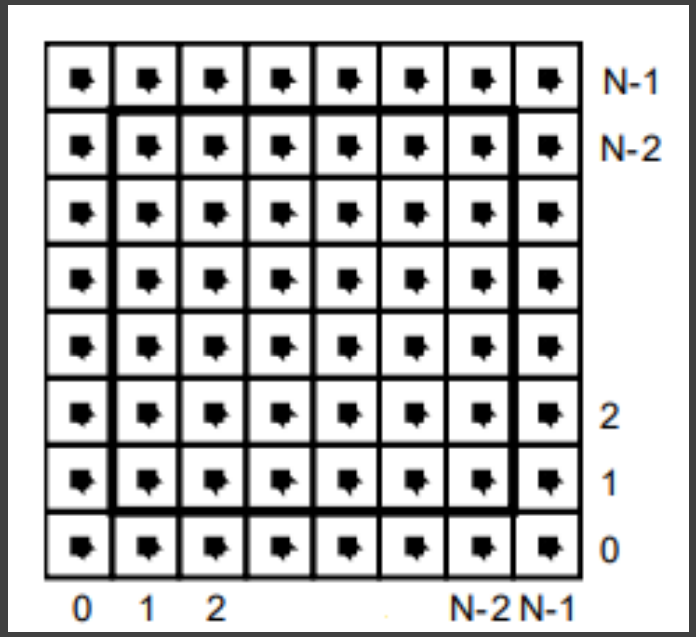
Navier-Stokes : $\rho\left[\frac{\partial\vec{V}}{\partial t}+(\vec{V}\cdot\vec{\nabla})\vec{V}\right]=-\vec{\nabla}\mathrm{P}+\rho\vec{g}+\eta\Delta\vec{V}$

Conservation de la masse : $\vec{\nabla}\cdot\ \vec{V}=0$

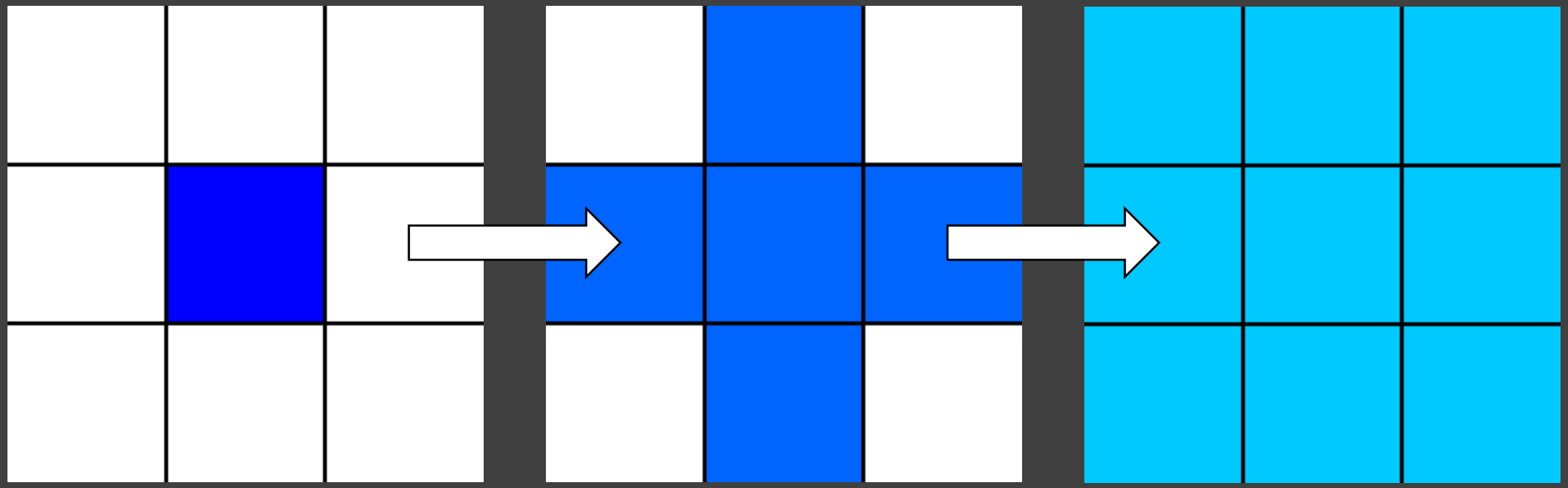
Méthode des éléments finis

- L'espace fini est divisé en cellules:
  - Chaque cellule « échange » de la matière avec ses voisines

- Principe de l'algorithme:
  - Diffusion
  - Mise à jour de la vitesse
  - Advection
  - Calcul de la force subie par la fusée



Diffuse

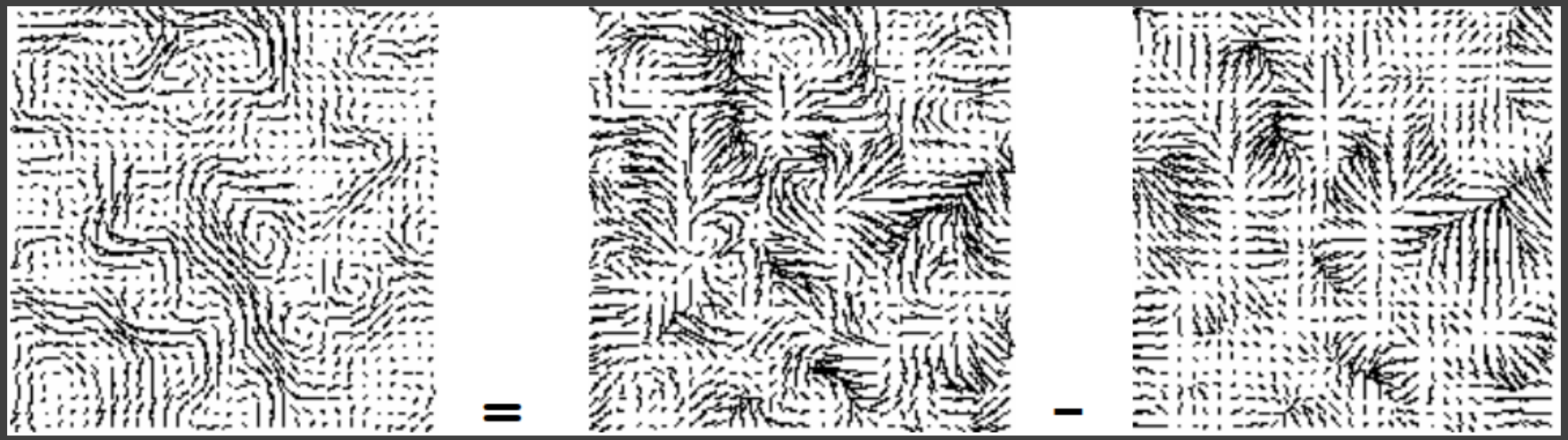


project

Récupère la partie incompressible du champ de vitesse

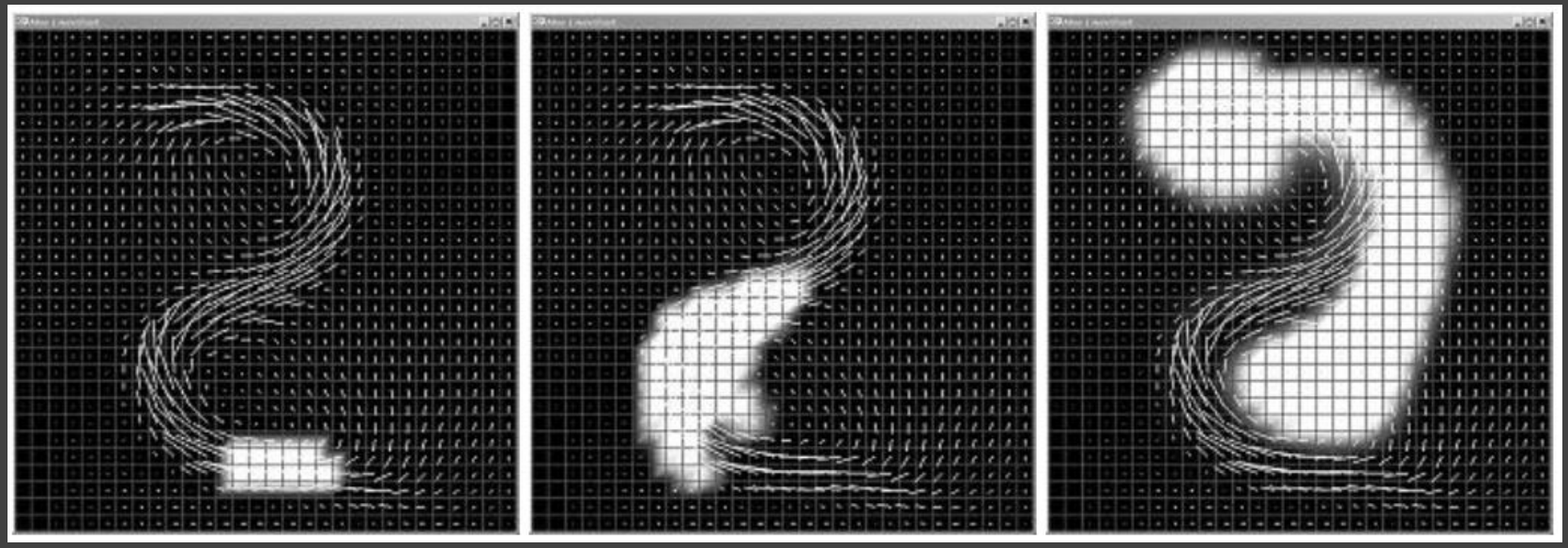
Décomposition de Hodge:

$$\overrightarrow{V_{new}} = \overrightarrow{V_{old}} - \overrightarrow{grad}(\overrightarrow{V_{old}})$$



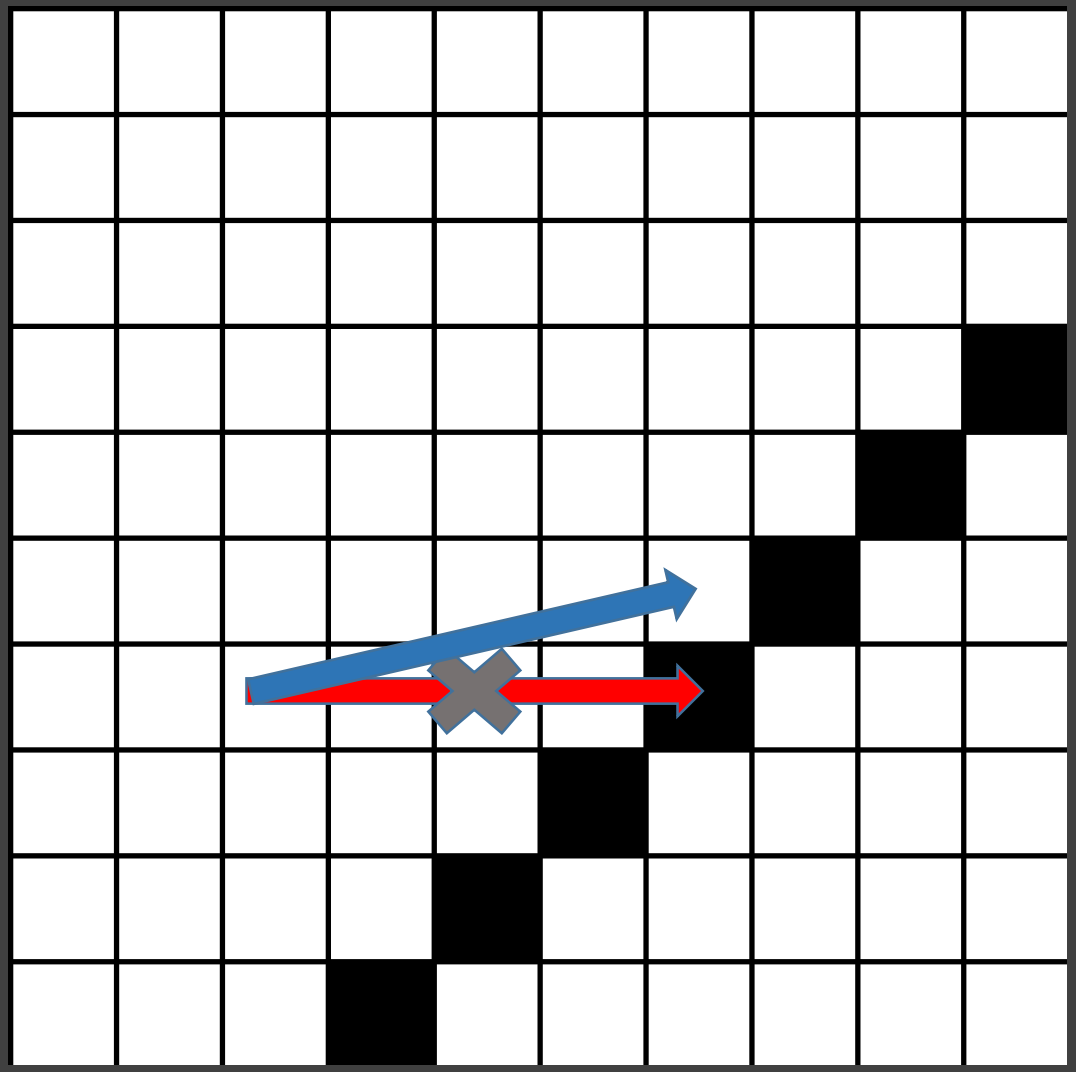
Real-Time Fluid Dynamics for Games, Jos Stam

Advect



Real-Time Fluid Dynamics for Games, Jos Stam

Calcul de la force



Problèmes de la simulation

- Très complexe temporellement et spatialement
- Difficile à mettre en place pour des fluides compressibles

http://www.nasa.gov/sites/default/files/thumbnails/image/42374725534_a653d1b7

Annexes

Codes en python et en C

```python
"""Modelisation de la trajectoire d'une fusée
    Méthode d'Euler contre solution de l'équation différentielle
"""

import matplotlib.pyplot as plt
from math import exp, pi


class Fusee:

    def __init__(self, m_vide, m_carburant, ve, dm):

        self.masse_vide = m_vide
        self.masse_carburant = m_carburant
        self.masse_initiale = m_vide + m_carburant
        self.masse = self.masse_initiale

        self.carburant = self.masse_carburant

        self.v = 0
        self.a = 0
        self.pos = 0

        self.poussee = ve*dm
        self.dm = dm
        self.ve = ve

    def update(self, dt):
        """Fonction de la méthode d'Euler"""

        self.a = 0

        if self.carburant > 0:

            self.carburant -= self.dm * dt
            self.masse -= self.dm*dt

            self.a += self.poussee


        # Le Cx évolue avec la vitesse, tableau suivant basé sur les données de la
        page wikipedia.com/en/Drag_(physics)

        list_cx = [(0.7 , 0.2),
                   (1, 1.4),
                   (1.1, 1.4),
                   (1.2, 0.9),
                   (1.3, 0.75),
                   (1.5, 0.7),
                   (1.7,0.5),
                   (2, 0.4)]

        c = self.v / 340
        Cx = 0.4
        for e in list_cx:
            if c <= e[0]:
                Cx = e[1]
                break

        rho = calcul_rho(self.pos)

        S = 6 * pi
        R = 1 / 2 * Cx * rho * S

        self.a -= pow(self.v, 2) * R

        # a = 1
        # self.a -= self.v* a

        self.a -= g * self.masse

        self.a /= self.masse

        self.v += self.a * dt
        self.pos += self.v * dt

def m(t, f):
    """Retourne la masse de la fusée à l'instant t"""

    return f.masse_initiale -f.dm*t

def h(t, f):
    """Retourne la hauteur de la fusée à l'instant t"""

    Mc = f.masse_carburant
    Dm = f.dm
    P = f.dm * f.ve
    a = 1
    m0 = f.masse_initiale

    A = -(g/(Dm-a)*m0 +P/a)*pow(m0, -a/Dm)

    z = lambda x: -g/(Dm-a)/(2*Dm) * pow(m(x,f), 2) +P/a*x - A/(a + Dm)*pow(m(x, f),
    a/Dm +1)

    return z(t) - z(0)

def h2(t, f, t0=180):
    """Retourne la hauteur de la fusée à l'instant t avec t=0 l'instant où les
    moteurs se coupent
    """
    a = 1
    M = m(t0, f)
    c = (v(t0, f) + M*g/a)

    z = lambda x: -M/a*c*exp(-a/M*(x-t0)) - M/a *g *(x-t0)

    return h(t0,f) + z(t) - z(t0)

def v(t, f):
    """Solution analytique de la vitesse dans la première phase du vol"""
    Dm = f.dm
    a = 1
    P = Dm * f.ve
    m0 = f.masse_initiale

    A = -(g/(Dm-a)*m0 +P/a)*pow(m0, -a/Dm)
    vit = lambda x: g/(Dm-a) * m(x, f) + P/a + A*pow(m(x, f), a/Dm)
    return vit(t) - vit(0)

def v2(t, f, t0=150):
    """Solution analytique de l'équa diff après qu'il n'y ait plus d'essence"""
    a = 1
    c = (v(t0, f) + m(t0, f)*g/a)
    m0 = m(t0, f)
    return c*exp(-a/m0 * (t-t0)) - m0/a *g

def calcul_rho(z):
    """Calcul la pression de l'air à une hauteur z en se basant sur le modèle de
    l'atmosphère isotherme"""

    R = 8.31
    T = 300
    M = 29e-3

    P = 10e5*exp(-(z*M*g)/(R*T))
    return M*P/(R*T)

g = 9.81
```

```python
m_vide = 130e3 + 481e3 + 107e3  # masse 1er étage à vide + charge max
m_carburant = 2300e3
t_b = 150
dm  = m_carburant/t_b

ve = 2500

# m_vide = 0.110
# m_carburant = 0.5
# t_b = 1
# dm = m_carburant / t_b

# ve = 30

fusee = Fusee(m_vide, m_carburant, ve, dm)

t_exec = 160
dt = 0.01
steps = int(t_exec / dt)

yl = list()
tl = list()
zl = [0]
t = 0

for i in range(steps):

    if fusee.pos >= 0:

        fusee.update(dt)
        yl.append(fusee.pos)

    else:
        yl.append(-0.1)

    if t < t_b:
        zl.append(h(t, fusee))

    else:
        temp = h2(t, fusee, t_b)
        zl.append(temp if temp >=0 else 0)

    tl.append(t)
    t += dt

plt.plot(tl, yl, label="Méthode d'Euler")
plt.plot(tl, zl[1:], label="Solution analytique")

plt.legend()

plt.xlabel("temps (s)")
plt.ylabel("hauteur (m)")

plt.show()
```

```
#ifndef DEF_FLUID

#define DEF_FLUID

struct FluidCube {
    int size;
    float dt;
    float diff;
    float visc;

    float *s;
    float *density;

    float *Vx;
    float *Vy;
    float *Vz;

    float *Vx0;
    float *Vy0;
    float *Vz0;
};
typedef struct FluidCube FluidCube;


FluidCube* FluidCubeCreate(int size, int diffusion, int viscosity, float dt);
void FluidCubeFree(FluidCube* cube);
void FluidCubeStep(FluidCube* cube, int fusee[], float* fx, float* fy, float* fz);
void FluidCubeAddDensity(FluidCube* cube, int x, int y, int z, float amount);
void FluidCubeAddVelocity(FluidCube* cube, int x, int y, int z, float amountX, float
amountY, float amountZ);
void force(int fusee[], float density[], float vx[], float vy[], float vz[], int N,
float dt, float* fx, float* fy, float* fz);


#endif

```

```c
#include <stdlib.h>
#include <math.h>
#include "fluid.h"

#define IX(x, y, z) ((x) + (y) * N + (z) * N * N)

static void set_bnd(int b, float x[], int N);
static void lin_solve(int b, float x[], float x0[], float a, float c, int iter, int N);
static void diffuse(int b, float x[], float x0[], float diff, float dt, int iter, int N);
static void advect(int b, float d[], float d0[], float volcX[], float velocY[], float velocZ[], float dt, int N);
static void project(float velocX[], float velocY[], float velocZ[], float p[], float div[], int iter, int N);


FluidCube *FluidCubeCreate(int size, int diffusion, int viscosity, float dt)
{
    FluidCube *cube = malloc(sizeof(*cube));
    int N = size;

    cube->size = size;
    cube->dt = dt;
    cube->diff = diffusion;
    cube->visc = viscosity;

    cube->s = calloc(N * N * N, sizeof(float));
    cube->density = calloc(N * N * N, sizeof(float));

    cube->Vx = calloc(N * N * N, sizeof(float));
    cube->Vy = calloc(N * N * N, sizeof(float));
    cube->Vz = calloc(N * N * N, sizeof(float));

    cube->Vx0 = calloc(N * N * N, sizeof(float));
    cube->Vy0 = calloc(N * N * N, sizeof(float));
    cube->Vz0 = calloc(N * N * N, sizeof(float));

    return cube;
}

void FluidCubeFree(FluidCube *cube)
{
    free(cube->s);
    free(cube->density);

    free(cube->Vx);
    free(cube->Vy);
    free(cube->Vz);

    free(cube->Vx0);
    free(cube->Vy0);
    free(cube->Vz0);

    free(cube);
}


static void set_bnd(int b, float x[], int N)
{
    for(int j = 1; j < N - 1; j++) {
        for(int i = 1; i < N - 1; i++) {
            x[IX(i, j, 0  )] = b == 3 ? -x[IX(i, j, 1  )] : x[IX(i, j, 1  )];
            x[IX(i, j, N-1)] = b == 3 ? -x[IX(i, j, N-2)] : x[IX(i, j, N-2)];
        }
    }
    for(int k = 1; k < N - 1; k++) {
        for(int i = 1; i < N - 1; i++) {
            x[IX(i, 0  , k)] = b == 2 ? -x[IX(i, 1  , k)] : x[IX(i, 1  , k)];
            x[IX(i, N-1, k)] = b == 2 ? -x[IX(i, N-2, k)] : x[IX(i, N-2, k)];
        }
    }
    for(int k = 1; k < N - 1; k++) {
        for(int j = 1; j < N - 1; j++) {
            x[IX(0  , j, k)] = b == 1 ? -x[IX(1  , j, k)] : x[IX(1  , j, k)];
            x[IX(N-1, j, k)] = b == 1 ? -x[IX(N-2, j, k)] : x[IX(N-2, j, k)];
        }
    }

    x[IX(0, 0, 0)]       = 0.33f * (x[IX(1, 0, 0)]
                                  + x[IX(0, 1, 0)]
                                  + x[IX(0, 0, 1)]);

    x[IX(0, N-1, 0)]     = 0.33f * (x[IX(1, N-1, 0)]
                                  + x[IX(0, N-2, 0)]
                                  + x[IX(0, N-1, 1)]);

    x[IX(0, 0, N-1)]     = 0.33f * (x[IX(1, 0, N-1)]
                                  + x[IX(0, 1, N-1)]
                                  + x[IX(0, 0, N)]);

    x[IX(0, N-1, N-1)]   = 0.33f * (x[IX(1, N-1, N-1)]
                                  + x[IX(0, N-2, N-1)]
                                  + x[IX(0, N-1, N-2)]);

    x[IX(N-1, 0, 0)]     = 0.33f * (x[IX(N-2, 0, 0)]
                                  + x[IX(N-1, 1, 0)]
                                  + x[IX(N-1, 0, 1)]);

    x[IX(N-1, N-1, 0)]   = 0.33f * (x[IX(N-2, N-1, 0)]
                                  + x[IX(N-1, N-2, 0)]
                                  + x[IX(N-1, N-1, 1)]);

    x[IX(N-1, 0, N-1)]   = 0.33f * (x[IX(N-2, 0, N-1)]
                                  + x[IX(N-1, 1, N-1)]
                                  + x[IX(N-1, 0, N-2)]);

    x[IX(N-1, N-1, N-1)] = 0.33f * (x[IX(N-2, N-1, N-1)]
                                  + x[IX(N-1, N-2, N-1)]
                                  + x[IX(N-1, N-1, N-2)]);
}

static void lin_solve(int b, float x[], float x0[], float a, float c, int iter, int N)
{
    float cRecip = 1.0 / c;
    for (int k = 0; k < iter; k++) {
        for (int m = 1; m < N - 1; m++) {
            for (int j = 1; j < N - 1; j++) {
                for (int i = 1; i < N - 1; i++) {
                    x[IX(i, j, m)] =
                        (x0[IX(i, j, m)]
                            + a*(    x[IX(i+1, j  , m  )]
                                    +x[IX(i-1, j  , m  )]
                                    +x[IX(i  , j+1, m  )]
                                    +x[IX(i  , j-1, m  )]
                                    +x[IX(i  , j  , m+1)]
                                    +x[IX(i  , j  , m-1)]
                                )) * cRecip;
                }
            }
        }
        set_bnd(b, x, N);
    }
}

static void diffuse (int b, float x[], float x0[], float diff, float dt, int iter, int N)
{
    float a = dt * diff * (N - 2) * (N - 2);
    lin_solve(b, x, x0, a, 1 + 6 * a, iter, N);
}

static void advect(int b, float d[], float d0[],  float velocX[], float velocY[], float velocZ[], float dt, int N)
```

```
{
    float i0, i1, j0, j1, k0, k1;

    float dtx = dt * (N - 2);
    float dty = dt * (N - 2);
    float dtz = dt * (N - 2);

    float s0, s1, t0, t1, u0, u1;
    float tmp1, tmp2, tmp3, x, y, z;

    float Nfloat = N;
    float ifloat, jfloat, kfloat;
    int i, j, k;

    for(k = 1, kfloat = 1; k < N - 1; k++, kfloat++) {
        for(j = 1, jfloat = 1; j < N - 1; j++, jfloat++) {
            for(i = 1, ifloat = 1; i < N - 1; i++, ifloat++) {

                // les variables tmp représente le déplacement suivant les trois
                directions de l'espace
                tmp1 = dtx * velocX[IX(i, j, k)];
                tmp2 = dty * velocY[IX(i, j, k)];
                tmp3 = dtz * velocZ[IX(i, j, k)];

                // représente l'indice de la case où doit se retrouver le fluide
                x    = ifloat - tmp1;
                y    = jfloat - tmp2;
                z    = kfloat - tmp3;

                // on s'assure de rester avec des indices valides pour le tableau
                if(x < 0.5) x = 0.5;
                if(x > Nfloat + 0.5) x = Nfloat + 0.5;

                // i1 représente la case voisine de i0, ce qui sert dans la suite du
                calcul en fonction de la partie décimale de x pour répartir la
                matière convenablement ce qui semble donner une direction
                préférentielle à la simulation ?
                i0 = floorf(x);
                    i1 = i0 + 1.0;

                if(y < 0.5) y = 0.5;
                    if(y > Nfloat + 0.5) y = Nfloat + 0.5;

                j0 = floorf(y);
                    j1 = j0 + 1.0;

                if(z < 0.5f) z = 0.5;
                    if(z > Nfloat + 0.5) z = Nfloat + 0.5;

                k0 = floorf(z);
                k1 = k0 + 1.0;

                s1 = x - i0;  // représente la partie décimale de x
                s0 = 1.0 - s1;
                t1 = y - j0;
                t0 = 1.0 - t1;
                u1 = z - k0;
                u0 = 1.0 - u1;

                int i0i = i0;
                int i1i = i1;
                int j0i = j0;
                int j1i = j1;
                int k0i = k0;
                int k1i = k1;

                // on répartit le fluide selon la partie décimale à laquelle il se
                trouve suivant les trois directions de l'espace.
                d[IX(i, j, k)] =

                    s0 * ( t0 * (u0 * d0[IX(i0i, j0i, k0i)]
                                +u1 * d0[IX(i0i, j0i, k1i)])
                        +( t1 * (u0 * d0[IX(i0i, j1i, k0i)]
                                +u1 * d0[IX(i0i, j1i, k1i)])))
                   +s1 * ( t0 * (u0 * d0[IX(i1i, j0i, k0i)]
                                +u1 * d0[IX(i1i, j0i, k1i)])
                        +( t1 * (u0 * d0[IX(i1i, j1i, k0i)]
                                +u1 * d0[IX(i1i, j1i, k1i)])));
            }
        }
    }
    set_bnd(b, d, N);
}

static void project(float velocX[], float velocY[], float velocZ[], float p[], float
div[], int iter, int N)
{
    //Les listes p et div ne servent qu'à stocker des données temporairement plutôt
    que de créer de nouvelles listes.
    for (int k = 1; k < N - 1; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {
                div[IX(i, j, k)] = -0.5*(
                          velocX[IX(i+1, j  , k  )]
                         -velocX[IX(i-1, j  , k  )]
                         +velocY[IX(i  , j+1, k  )]
                         -velocY[IX(i  , j-1, k  )]
                         +velocZ[IX(i  , j  , k+1)]
                         -velocZ[IX(i  , j  , k-1)]
                    )/N;
                p[IX(i, j, k)] = 0;
            }
        }
    }
    set_bnd(0, div, N);
    set_bnd(0, p, N);
    lin_solve(0, p, div, 1, 6, iter, N);

    for (int k = 1; k < N - 1; k++) {
        for (int j = 1; j < N - 1; j++) {
            for (int i = 1; i < N - 1; i++) {
                velocX[IX(i, j, k)] -= 0.5 * (  p[IX(i+1, j, k)]
                                                -p[IX(i-1, j, k)]) * N;
                velocY[IX(i, j, k)] -= 0.5 * (  p[IX(i, j+1, k)]
                                                -p[IX(i, j-1, k)]) * N;
                velocZ[IX(i, j, k)] -= 0.5 * (  p[IX(i, j, k+1)]
                                                -p[IX(i, j, k-1)]) * N;
            }
        }
    }
    set_bnd(1, velocX, N);
    set_bnd(2, velocY, N);
    set_bnd(3, velocZ, N);
}

void FluidCubeStep(FluidCube *cube, int fusee[], float* fx, float* fy, float* fz)
{
    int N          = cube->size;
    float visc     = cube->visc;
    float diff     = cube->diff;
    float dt       = cube->dt;
    float *Vx      = cube->Vx;
    float *Vy      = cube->Vy;
    float *Vz      = cube->Vz;
    float *Vx0     = cube->Vx0;
    float *Vy0     = cube->Vy0;
    float *Vz0     = cube->Vz0;
    float *s       = cube->s;
    float *density = cube->density;

    // la vitesse d'une cellule se propage à ses voisines à cause de la viscosité du
    fluide
    diffuse(1, Vx0, Vx, visc, dt, 4, N);
    diffuse(2, Vy0, Vy, visc, dt, 4, N);
```

```
275         diffuse(3, Vz0, Vz, visc, dt, 4, N);
276
277         force(fusee, s, Vx, Vy, Vz, N, dt, fx, fy, fz);
278
279         // les listes Vx et Vy ne sont utilisées qu'à des fins de stockage temporaire
280         project(Vx0, Vy0, Vz0, Vx, Vy, 4, N);
281
282         // le fluide se déplace en conservant sa vitesse donc on déplace le champ de
            vitesse en conséquence
283         advect(1, Vx, Vx0, Vx0, Vy0, Vz0, dt, N);
284         advect(2, Vy, Vy0, Vx0, Vy0, Vz0, dt, N);
285         advect(3, Vz, Vz0, Vx0, Vy0, Vz0, dt, N);
286
287         project(Vx, Vy, Vz, Vx0, Vy0, 4, N);
288
289         diffuse(0, s, density, diff, dt, 4, N);
290         advect(0, density, s, Vx, Vy, Vz, dt, N);
291     }
292
293     void FluidCubeAddDensity(FluidCube *cube, int x, int y, int z, float amount)
294     {
295         int N = cube->size;
296         cube->density[IX(x, y, z)] += amount;
297     }
298
299     void FluidCubeAddVelocity(FluidCube *cube, int x, int y, int z, float amountX, float
    amountY, float amountZ)
300     {
301         int N = cube->size;
302         int index = IX(x, y, z);
303
304         cube->Vx[index] += amountX;
305         cube->Vy[index] += amountY;
306         cube->Vz[index] += amountZ;
307     }
308
309
310     void force(int fusee[], float density[], float vx[], float vy[], float vz[], int N,
    float dt, float* fx, float* fy, float* fz){
311
312         // on suppose que dt est assez petit pour que le fluide ne rentre pas dans la
            fusee
313         // et que le déplacement du fluide est selon l'axe OX
314
315         float dtr = dt * (N-2);
316         float voisins = 0;
317         for(int i = 1; i < N-1; i++){
318             for(int j = 1; j < N-1; j++){
319                 for(int k = 1; k < N-1; k++){
320
321                     int x = i + vx[IX(i, j, k)] * dtr;
322                     int y = j + vy[IX(i, j, k)] * dtr;
323                     int z = k + vz[IX(i, j, k)] * dtr;
324
325                     // s'il y a collision on déplace le fluide
326                     if(fusee[IX(x, y, z)]){
327                         voisins = 0;
328                         // on compte le nombre de voisins libres autour de la case occupée
329                         for(int m = -1; m <= 1; m++){
330                             for(int l = -1; l <= 1; l++){
331                                 if(!fusee[IX(x, y+m, z+l)])
332                                     voisins++;
333                             }
334                         }
335                     }
336                     // pour chaque voisins non occupé on envoie la même quantité de
                        matière
337                     for(int m = -1; m <= 1; m++){
338                         for(int l = -1; l <= 1; l++){
339                             if(!fusee[IX(x, y+m, z+l)]){
340                                 float d  = density[IX(i, j, k)] / voisins;
341                                 float d0 = density[IX(x, y+m, z+l)];
```



```
342
343                             // TODO changer la vitesse en lui appliquant deux
                                rotations
344                             // on conserve la norme de la vitesse
345
346                             float old_vx = vx[IX(i, j, k)];
347                             float old_vy = vy[IX(i, j, k)];
348                             float old_vz = vz[IX(i, j, k)];
349
350                             // Reste à faire la rotation de la vitesse
351                             float new_vx = 0;
352                             float new_vy = 0;
353                             float new_vz = 0;
354
355                             // on fait la moyenne des vitesses
356                             vx[IX(x, y+m, z+l)] = d*vx[IX(i, j, k)] + d0*vx[IX(x,
                                y+m, z+l)];
357
358                             // on ajoute la quantité de matière adéquate
359                             density[IX(x, y+m, z+l)] += d;
360                             density[IX(i, j, k)] = 0; // on déplace toute la matière
                                de la case d'origine
361
362                             // et calculer la norme de la force (f = m*dv/dt)
363                             float rho = 10e-3;
364                             *fx += rho * d * (new_vx - old_vx) / dt;
365                             *fy += rho * d * (new_vy - old_vy) / dt;
366                             *fz += rho * d * (new_vz - old_vz) / dt;
367
368                         }
369                     }
370                 }
371             }
372         }
373     }
374 }
375
```

```c
#include <stdlib.h>
#include <stdio.h>

#include "fluid.h"

#define IX(x, y, z) ((x) + (y) * SIZE + (z) * SIZE * SIZE)
#define SIZE 256


void initialisation(FluidCube* fluid, int fusee[]);

int main(int argc, char** argv)
{
    int *fusee = calloc(SIZE*SIZE*SIZE, sizeof(int));
    FluidCube* fluid = FluidCubeCreate(SIZE, 1, 1, 0.1);

    float fx, fy, fz;
    FluidCubeAddDensity(fluid, SIZE/2, SIZE/2, SIZE/2, 100);
    FluidCubeStep(fluid, fusee, &fx, &fy, &fz);
    /*
    for(int i = 0; i < SIZE; i++){
        for(int j = 0; j < SIZE; j++){
            printf("%f ", fluid->density[i + j * SIZE + SIZE/2]);
        }
    }
    */
    //printf("%f\n", fx);
    char var = ' ';
    printf("fini !\nAppuyez sur n'importe quelle touche pour quitter... ");
    scanf("%c", &var);

    FluidCubeFree(fluid);
    free(fusee);

    return 1;
}

void initialisation(FluidCube* fluid, int fusee[])
{
    int N = fluid->size;
    float* density = fluid->s;

    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            for(int k = 0; k < N; k++){
                density[IX(i,j,k)] = 10;
                fusee[IX(i, j, k)] = 0;
            }
        }
    }
}

```