# Simulation numérique de vêtements par le système Masses - Ressorts

Abdeladim BOUDERSA

Numéro CNC : MK114M

Session 2024

# Introduction

**(1)** Modélisation d'un T-Shirt

**(2)** Mise en avant des plis sur un tissu

**(3)** Maillage d'un modèle 3D

**Figure 1:** Exemples de modélisation numérique d'un tissu

# Présentation du système

On modélise le système masses-ressorts comme suit :

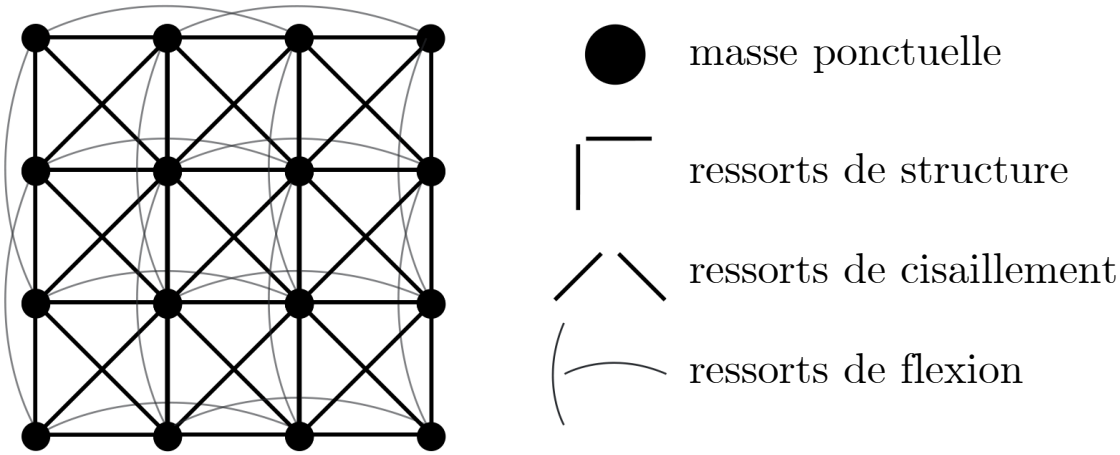



**Figure 2:** Présentation du modèle Masses-Ressorts

## Problématique:

Le modèle masses-ressorts permet-t-il de représenter le comportement mécanique du tissu de manière réaliste en considérant les différentes contraintes auxquelles celui-ci est soumis? Peut-on simuler numériquement ce système ?

# Table de Matière

# Schématisation du système à une masse



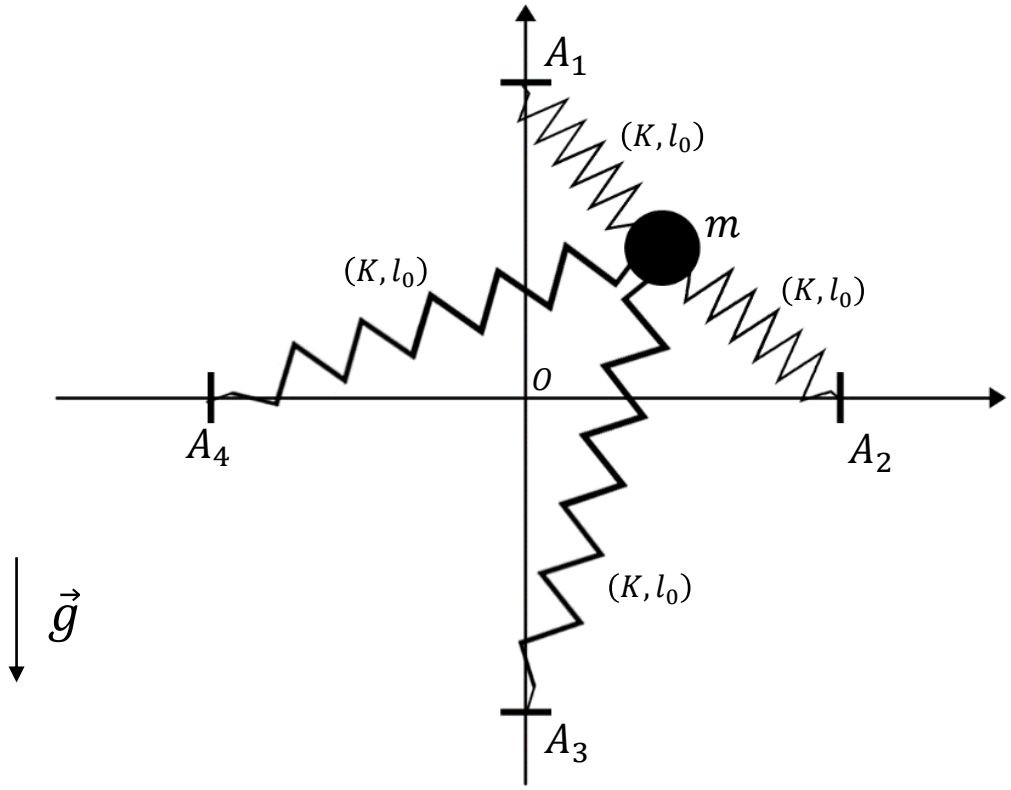



**Figure 3:** Modèle simplifié d'un système à une masse

# Le comportement dynamique du système à une masse

- Le Bilan de forces :

- $\overrightarrow{F_i} = -K\left(\overrightarrow{A_iM} - \overrightarrow{l_0}\right)$ : Force élastique exercée par le ressort $i$, de constante de raideur $K$, et de longueur à vide $l_0$.
- $\vec{f} = -C_{amort}\frac{d\overrightarrow{OM}}{dt}$ : Force d'amortissement visqueux de coefficient d'amortissement $C_{amort}$.
- $\vec{P} = -mg\vec{y}$: Poids de la particule de masse $m$.

Où : $\overrightarrow{OM} = x\overrightarrow{e_x} + y\overrightarrow{e_y} + z\overrightarrow{e_z}$

# Equation différentielle du mouvement

Soit $p \in \{x, y, z\}$

- On applique le P.F.D au système, on obtient l'équation différentielle suivante :

$$\boxed{\ddot{p} + \frac{C_{amort}}{m}\dot{p} + \frac{4k}{m}p = \begin{cases} 0 & ,si\ p = x \\ -g & ,si\ p = y \\ 0 & ,si\ p = z \end{cases}}$$

On pose : $\theta = \begin{pmatrix} \dot{p} \\ p \end{pmatrix}$

- On retrouve une équation différentielle de 1er ordre:

$$\boxed{\dot{\theta} = A.\theta} \quad (1)$$

Où: $A = \begin{pmatrix} -\frac{C_{amort}}{m} & -\frac{4k}{m} \\ 1 & 0 \end{pmatrix}$

On résout l'équation différentielle (1) pour chacune des 3 coordonnées, et on trace avec python :

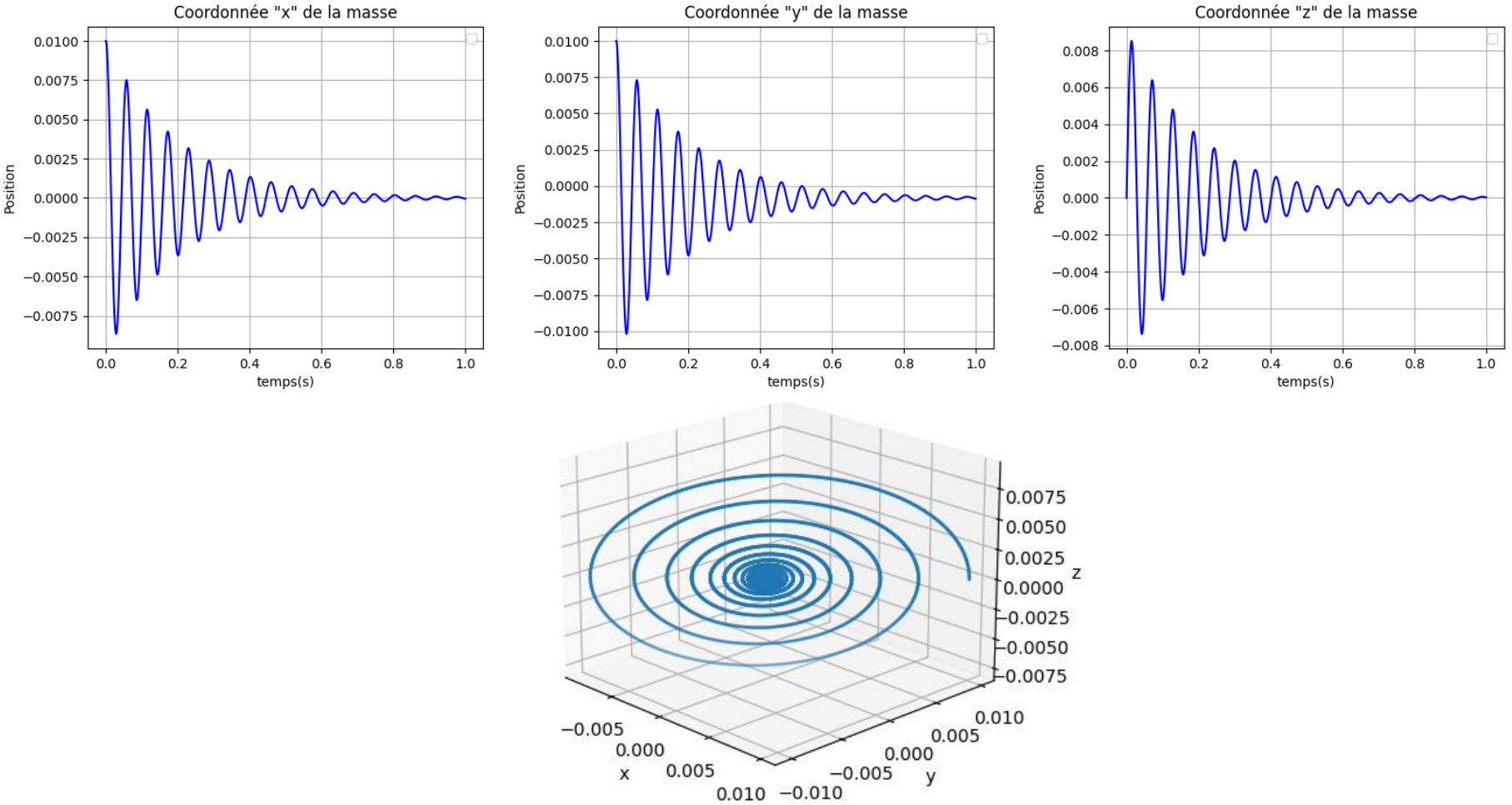



**Figure 4:** Courbes représentant la position spatiale de la masse en fonction du temps

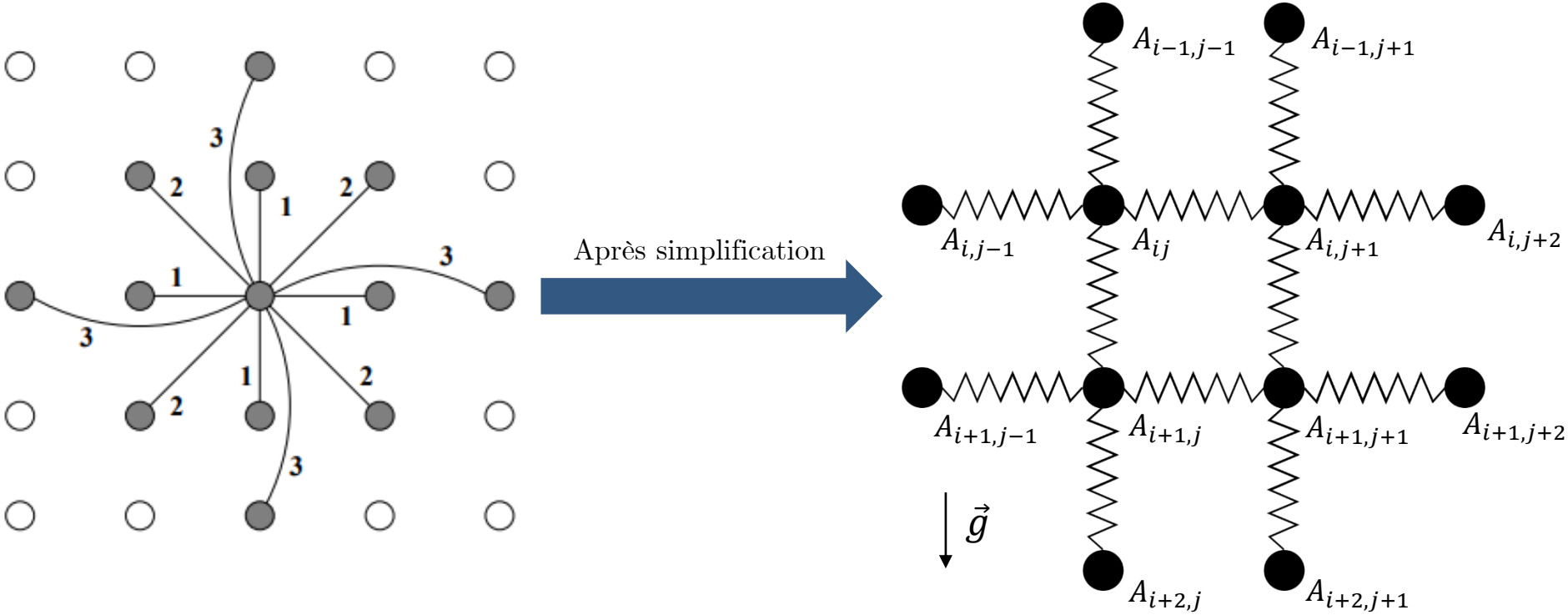



**Figure 5:** Simplification du modèle

- Considérons le système { masse $A_{ij}$}
- Le Bilan de forces :

➢ $\overrightarrow{F_{ijkl}} = -\sum_{(k,l)\in\beta} K_{i,j,k,l}(\overrightarrow{A_{ij}A_{kl}} - l_{0_{i,j,k,l}}\overrightarrow{u_{ij}})$ : Résultante des forces élastiques appliquées par les ressorts voisins, où:

- $\beta$ : est l'ensemble des couple $(k,l)$ tels que $A_{kl}$ est voisin de $A_{ij}$.
- $l_{0_{i,j,k,l}}$: longueur à vide du ressort $R_{i,j,k,l}$.
- $K_{i,j,k,l}$: Constante de raideur du ressort $R_{i,j,k,l}$.
- $\overrightarrow{u_{ijkl}}$: un vecteur unitaire, tel que: $\overrightarrow{u_{ij}} = \frac{\overrightarrow{A_{ij}A_{kl}}}{\|\overrightarrow{A_{ij}A_{kl}}\|}$

➢ $\overrightarrow{f_{ij}} = -C_{amort}\overrightarrow{v_{ij}}$: Force d'amortissement visqueux de coefficient d'amortissement $C_{amort}$, où $\overrightarrow{v_{ij}}$ est le vecteur vitesse de $A_{ij}$

➢ $\overrightarrow{P_{ij}} = -m_{ij}\, g\vec{y}$: Poids de la particule de masse $m_{ij}$ .

- On applique la 2ème loi de Newton:

$$\ddot{p}_{ij} + \frac{C_{amort}}{m}\dot{p}_{ij} + \frac{k}{m}\sum_{(k,l)\in\beta}\left(1 - \frac{l_{0_{i,j,k,l}}}{\|p_{ij} - p_{kl}\|}\right).(p_{ij} - p_{kl}) = -g\begin{pmatrix}0\\1\\0\end{pmatrix}$$

où: $p_{ij} = \begin{pmatrix}x_{ij}\\y_{ij}\\z_{ij}\end{pmatrix}$

- On résout ces équations différentielles couplées par la méthode d'Euler explicite, adaptée au second ordre:

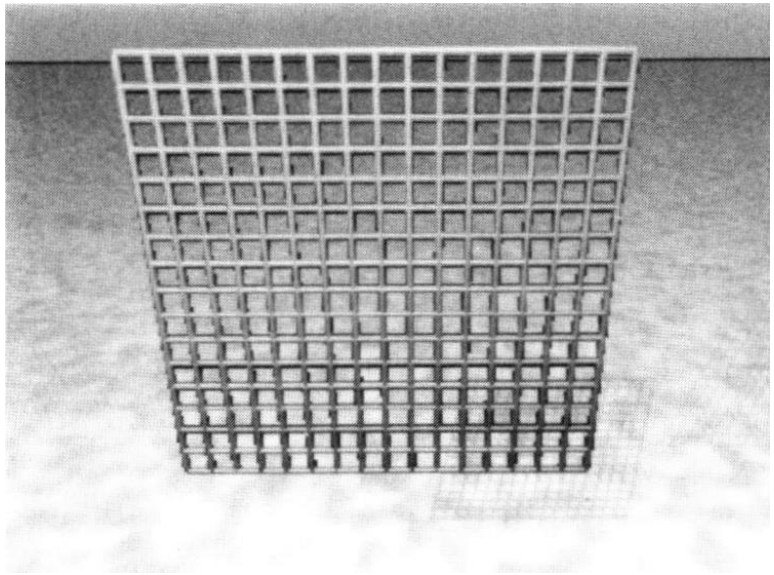
$$\begin{cases}\ddot{p}_{ij}(t) = \dfrac{1}{m_{ij}} f(t, \dot{p}_{ij}(t), p_{ij}(t))\\ \dot{p}_{ij}(t + \Delta t) = \dot{p}_{ij}(t) + \Delta t.\ddot{p}_{ij}(t)\\ p_{ij}(t + \Delta t) = p_{ij}(t) + \Delta t.\dot{p}_{ij}(t)\end{cases}$$

# Table de Matière

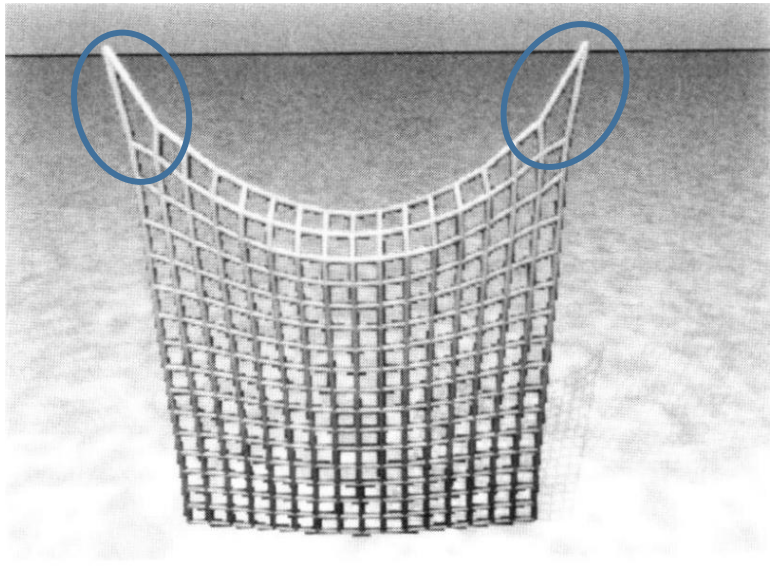
# Effet de "Super-Elasticité"



Mise en avant de l'effet:



(1) Position initiale



(1) Position après plusieurs itérations

**Figure 6:** Mise en avant de l'effet de "Super-Elasticité"

On constate une élongation importante des ressorts de cisaillement ainsi que ceux de Structure, ce qui diminue le réalisme de la simulation

# Solution proposée à ce phénomène



On peut le résoudre en changeant l'un des deux facteurs : $C_{amort}$ ou $K$, cependant chacune de ces deux solutions a des avantages et des inconvénients :

| | **Avantage** | **Inconvénient** |
|---|---|---|
| $C_{amort}$ | Corrige les oscillations non voulues, et augmente la stabilité | Augmente la rigidité du système de façon indésirable |
| $K$ | Diminue le taux de déformation | Nécessite une augmentation de précision et du coût |

# Solution proposée à ce phénomène

- La modification de la constante de raideur semble préférable et plus au service du réalisme de la simulation.

- En effet:

On sait que: $$\left\|\vec{F}\right\| = K.\Delta l \Rightarrow \quad l = \frac{\left\|\vec{F}\right\|}{K} + l_0$$

Et, on a: $$T_0 = 2\pi\sqrt{\frac{m}{K}} \quad \Rightarrow K = \frac{4\pi^2 m}{{T_0}^2}$$

On constate que:

- si $K$ augmente, alors $\Delta l$ diminue.
- si $K$ augmente, alors $T_0$ diminue, et par suite $\Delta t$ aussi.

Le choix de la modification de $K$ permet de corriger la déformation, cependant il nécessite plus d'itérations et donc il y'a augmentation du coût de l'algorithme.

# Phénomène de cisaillement isosurfacique

On note $\pi$ le plan $(Oxy)$

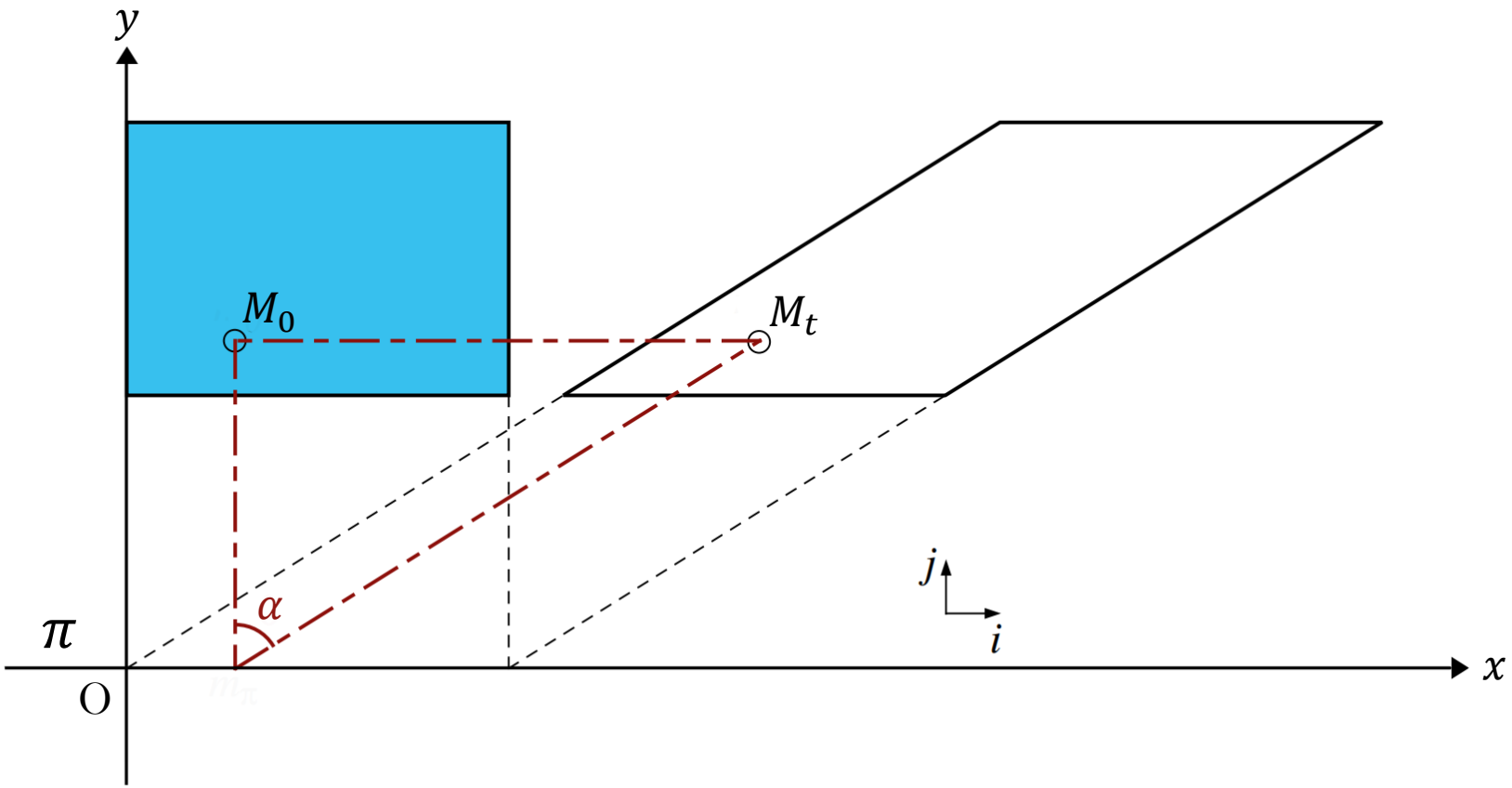



**Figure 7:** Schématisation du phénomène

- On définit le produit tensoriel de deux vecteurs comme suit :

$$\begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} \otimes \begin{pmatrix} Y_1 \\ Y_2 \\ Y_3 \end{pmatrix} = \begin{pmatrix} X_1Y_1 & X_1Y_2 & X_1Y_3 \\ X_2Y_1 & X_2Y_2 & X_2Y_3 \\ X_3Y_1 & X_3Y_2 & X_3Y_3 \end{pmatrix}$$

- On définit:
  - $K_t = \frac{\|d\overrightarrow{OM_t}\|}{\|d\overrightarrow{OM_0}\|}$: La dilatation linéique
  - $K_s = \frac{\|d\overrightarrow{S_t}\|}{\|d\overrightarrow{S_0}\|}$: La dilatation surfacique

**Expression du déplacement élémentaire:**

- On note $\Psi$ la translation de rapport $\sigma = \tan(\alpha)$, et de direction $\vec{\imath}$, du plan telle que :

$$\Psi(\overrightarrow{OM_0}) = \overrightarrow{OM_t}$$

On a : $$\overrightarrow{OM_t} = \overrightarrow{OM_0} + \sigma\left[\left(\overrightarrow{OM_0} - \overrightarrow{OM_\pi}\right).\ \vec{j}\right]\vec{\imath}$$

On a : $$\boxed{\overrightarrow{OM_t} = \overrightarrow{OM_0} + \sigma\left[\overrightarrow{OM_0}.\ \vec{j}\right]\vec{\imath}} \quad (2)$$

En différenciant (2), on obtient:

$$\Rightarrow d\overrightarrow{OM_t} = d\overrightarrow{OM_0} + \sigma\left[d\overrightarrow{OM_0}.\ \vec{j}\right]\vec{\imath}$$

$$\Rightarrow \boxed{d\overrightarrow{OM_t} = d\overrightarrow{OM_0} + \sigma\underbrace{(\vec{\imath} \otimes \vec{j})}_{\text{Démonstration en annexe}} d\overrightarrow{OM_0}} \quad (3)$$

**Propriétés de la transformation:**

D'après (3), on note :

$$F = I_3 + \sigma(\vec{i} \otimes \vec{j}) = \begin{pmatrix} 1 & \sigma & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

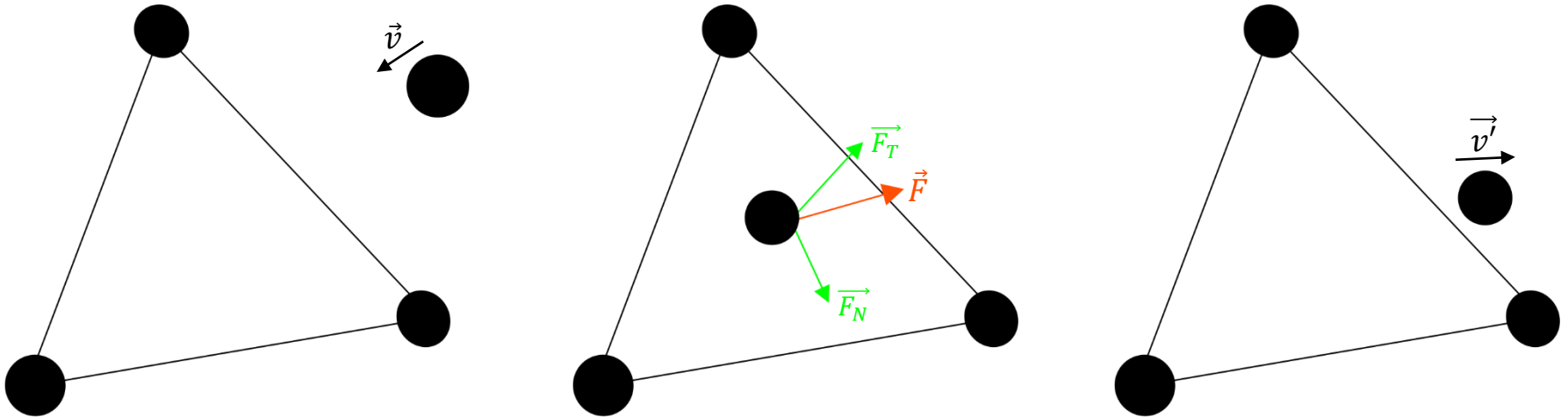
On a:

- $K_t = \frac{\|d\overrightarrow{OM_t}\|}{\|d\overrightarrow{OM_0}\|} = \|\overrightarrow{u_0} + \sigma(\overrightarrow{u_0}.\ \vec{j}).\vec{i}\|$ , où $\overrightarrow{u_0} = \frac{d\overrightarrow{OM_0}}{\|d\overrightarrow{OM_0}\|}$ :
  - Si $\overrightarrow{u_0} = \vec{i}$, alors: $\boxed{K_t = 1}$
  - Si $\overrightarrow{u_0} = \vec{j}$, alors: $\boxed{K_t = \sqrt{1+\sigma^2}}$
- $K_s = \underbrace{\frac{\|d\overrightarrow{S_t}\|}{\|d\overrightarrow{S_0}\|} = \det F}_{\text{Démonstration en annexe}}$, donc: $\boxed{K_s = 1}$

On conclut que :

- Le cisaillement isosurfacique conserve bien l'aire de la surface étudiée ainsi que l'une des 2 directions du plan de la surface.

- Ce phénomène est pris en considération dans le système masses-ressorts par le biais des ressorts de cisaillement.




**(1)** Avant le choc **(2)** Au moment du choc **(3)** Après le choc

**Figure 7:** schématisation du choc

On note:

- $\vec{N}$ la normale au triangle ABC au point de contact H
- $\vec{F} = \overrightarrow{F_T} + \overrightarrow{F_N}$: La force de contact appliquée sur le point H
- $\vec{v}$: la vitesse de P avant le choc
- $\overrightarrow{v'}$: la vitesse de P après le choc

**Lois de Coulomb:**

- Si $\|\overrightarrow{F_T}\| \geq k_f \|\overrightarrow{F_N}\|$, c'est-à-dire qu'il y'a une vitesse de glissement, et un frottement, la force appliquée sur P est donc :

$$\boxed{\overrightarrow{F_s} = \overrightarrow{F_T} - k_f \|\overrightarrow{F_N}\| . \overrightarrow{u_T}} \quad (4) \quad , \quad \text{où } \overrightarrow{u_T} = \frac{\overrightarrow{F_T}}{\|\overrightarrow{F_T}\|}$$

- Si $\|\overrightarrow{F_T}\| \leq k_f \|\overrightarrow{F_N}\|$, il n y'a pas de glissement, donc: $\overrightarrow{F_s} = 0$

**Expression de la vitesse après le choc :**

On fait l'approximation suivante : $\vec{F} \propto \vec{v}$, donc aussi: $\overrightarrow{F_s} \propto \overrightarrow{v_T}'$

⇒ On peut donc remplacer $\overrightarrow{F}$ par $\vec{v}$ dans les lois précédentes

- Il y'a dissipation d'énergie qui se manifeste à travers le coefficient de dissipation $k_d \in [0,1]$, telle que :

$$\overrightarrow{v'} = \overrightarrow{v_T}' - k_d \overrightarrow{v_N}$$

D'après (4) : $\overrightarrow{v_T}' = \overrightarrow{v_T} - k_f \|\overrightarrow{v_N}\| . \overrightarrow{u_T}$

Ainsi:

$$\begin{cases} \text{Si } \|\overrightarrow{v_T}\| \geq k_f \|\overrightarrow{v_N}\|, & \overrightarrow{v'} = \overrightarrow{v_T} - k_f \|\overrightarrow{v_N}\| . \overrightarrow{u_T} - k_d \overrightarrow{v_N} \\ \text{Si } \|\overrightarrow{v_T}\| \leq k_f \|\overrightarrow{v_N}\|, & \overrightarrow{v'} = -k_f \overrightarrow{v_N} \end{cases}$$

# Table de Matière

# Simulation du modèle masses-ressorts

(1) Exemples de Mouvement du tissu

(2) Collision du tissu avec la balle

**Figure 8:** Résultat de la simulation d'un tissu de taille 128x128 par python

# Tableau de Matière

# Conclusion



- Le système Masses-Ressorts représente bien le comportement physique le tissu et les contraintes qu'il subit.
- Il permet de ramener l'étude d'un milieu continu complexe à un milieu discret plus simple.
- Ce modèle est l'un des plus faciles à implémenter numériquement. Malgré ses limitations, c'est un compromis entre rapidité et justesse physique.

# MERCI POUR VOTRE ATTENTION

# Annexe 1: code python des 4 courbes



```python
import matplotlib.pyplot as plt
import scipy.integrate as sp
import numpy as np
#On définit les constantes
m=1e-3
g=9.8
k=3
c=1e-2

#On définit les différents équations différentielles
def F(x,t):
    return [(-1/m)*(c*float(x[0])+(4*k)*float(x[1])),float(x[0])]
def G(x,t):
    return [(-1/m)*(c*float(x[0])+(4*k)*float(x[1]))-g,float(x[0])]

#résolution des E.D par la méthode odeint de scipy.integrate
t=np.linspace(0,1,10000)
L=sp.odeint(F,[0,1e-2],t)
M=sp.odeint(G,[0,1e-2],t)
N=sp.odeint(F,[1,0],t)
positionx = [L[i][1] for i in range(len(L))]
positiony = [M[i][1] for i in range(len(M))]
positionz = [N[i][1] for i in range(len(N))]

#traçage des 3 courbes
plt.subplot(131)
plt.grid()
plt.xlabel('temps(s)')
plt.ylabel('Position')
plt.title('Coordonnée "x" de la masse')
plt.plot(t,positionx,'b-')
plt.legend()

plt.subplot(132)
plt.grid()
plt.xlabel('temps(s)')
plt.ylabel('Position')
plt.title('Coordonnée "y" de la masse')
plt.plot(t,positiony,'b-')
plt.legend()

plt.subplot(133)
plt.grid()
plt.xlabel('temps(s)')
plt.ylabel('Position')
plt.title('Coordonnée "z" de la masse')
plt.plot(t,positionz,'b-')
plt.legend()

plt.show()
```

```python
#traçage de la courbe en 3D
ax=plt.axes(projection='3d')
plt.grid()
ax.scatter(positionx,positiony,positionz,s=1)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.title('Coordonnée "z" de la masse')
plt.legend()

plt.show()
```

# Annexe 2: démonstration de la relation p.20



Soit $\vec{v}$ un vecteur, on a:

$$(\vec{i} \otimes \vec{j}).\overrightarrow{OM_0} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.\overrightarrow{OM_0} = \begin{pmatrix} \overrightarrow{OM_0}.\vec{j} \\ 0 \\ 0 \end{pmatrix}$$

$$(\overrightarrow{OM_0}.\vec{j})\vec{i} = \begin{pmatrix} \overrightarrow{OM_0}.\vec{j} \\ 0 \\ 0 \end{pmatrix}$$

# Annexe 3: démonstration de la relation p.21



On considère deux déplacements élémentaires de $M_0$: $\overrightarrow{dOM_0} = \begin{pmatrix} x_1 \\ x_2 \\ 0 \end{pmatrix}$ et $\overrightarrow{dOM_0}' = \begin{pmatrix} y_1 \\ y_2 \\ 0 \end{pmatrix}$

On a pour : $F = \begin{pmatrix} 1 & \sigma & 0 \\ 0 & t & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Donc :

$$\begin{aligned}
\overrightarrow{dS_t} &= \overrightarrow{dOM_t} \wedge \overrightarrow{dOM_t}' \\
&= F.\overrightarrow{dOM_0} \wedge F.\overrightarrow{dOM_0}' \\
&= \begin{pmatrix} x_1 + \sigma x_2 \\ t x_2 \\ 0 \end{pmatrix} \wedge \begin{pmatrix} y_1 + \sigma y_2 \\ t y_2 \\ 0 \end{pmatrix} \\
&= \begin{pmatrix} 0 \\ 0 \\ t(x_1 y_2 - x_2 y_1) \end{pmatrix} \\
&= t \begin{pmatrix} 0 \\ 0 \\ (x_1 y_2 - x_2 y_1) \end{pmatrix} \\
&= detF.\overrightarrow{dOM_0} \wedge \overrightarrow{dOM_0}' \\
&= detF.\overrightarrow{dS_0}
\end{aligned}$$

# Annexe 3: Code complet de la simulation



```python
import taichi as ti
ti.init(arch=ti.cuda)
#cette caractéristique de taichi lui permet de lire le code par le biais du gpu

n = 128
quad_size = 1.0 / n #taille d'un carré contenant 4 masses
dt = 4e-2 / n #pas temporel
substeps = int(1 / 100 // dt)

#On définit les constantes physiques
gravité = ti.Vector([0, -9.8, 0])
ressort_Y = 3e4
amortissement_relatif = 1e4 #due au mvt relative entre 2 masses
amortissement_visq = 1 #due à l'amortissement comme étudié précédemment

#Création de la balle
rayon = 0.3
centre = ti.Vector.field(3, dtype=float, shape=(1, ))
centre[0] = [0, 0, 0]

#x (resp v) est un champ 2D de taille n x n dont les éléments sont des vecteurs
#de taille 3 représentants les positions (resp.les vitesses) des masses ponc.
x = ti.Vector.field(3, dtype=float, shape=(n, n))
v = ti.Vector.field(3, dtype=float, shape=(n, n))

num_triangles = (n - 1) * (n - 1) * 2
#champ 1D stockant les indices des sommets pour construire le maillage.
indices = ti.field(int, shape=num_triangles * 3)
#champ 1D stockant les coordonnées des sommets pour construire le maillage.
coord = ti.Vector.field(3, dtype=float, shape=n * n)
couleurs = ti.Vector.field(3, dtype=float, shape=n * n)

ressorts_flexion = False
##
```

```python
##
#Initialisation du maillage
@ti.kernel
def initialisation_masses():

#on centre les masses au dessus du ballon :
    for i, j in x:
        x[i, j] = [
            i * quad_size - 0.5 , 0.6,
            j * quad_size - 0.5
        ]
        v[i, j] = [0, 0, 0]


@ti.kernel
def initialisation_indices():
    for i, j in ti.ndrange(n - 1, n - 1):
        carre_id = (i * (n - 1)) + j
        # 1er triangle du carré
        indices[carre_id * 6 + 0] = i * n + j
        indices[carre_id * 6 + 1] = (i + 1) * n + j
        indices[carre_id * 6 + 2] = i * n + (j + 1)
        # 2eme triangle du carré
        indices[carre_id * 6 + 3] = (i + 1) * n + j + 1
        indices[carre_id * 6 + 4] = i * n + (j + 1)
        indices[carre_id * 6 + 5] = (i + 1) * n + j

    for i, j in ti.ndrange(n, n):
            couleurs[i * n + j] = (0, 0.334, 0.92)

initialisation_indices()
##
```

# Annexe 3: Code complet de la simulation



```python
##
#une liste qui contient les indice des masses voisines de (0,0), il y'en a 12
#si les ressorts de flexion sont pris en compte alors il y'a 12 masses voisines
#sinon il y'en a 9
indices_vois = []
if ressorts_flexion:
    for i in range(-1, 2):
        for j in range(-1, 2):
            if (i, j) != (0, 0):
                indices_vois.append(ti.Vector([i, j]))

else:
    for i in range(-2, 3):
        for j in range(-2, 3):
            if (i, j) != (0, 0) and abs(i) + abs(j) <= 2:
                indices_vois.append(ti.Vector([i, j]))

#cette fonction permet d'exprimer l'effet des 4 facteur sur la position et la
#vitesse de chaque masse ponctuelle
@ti.kernel
def substep():
    #1.poids de chaque masse
    #on traverse la liste X comme un champ 1D ou une liste de taille nxm
    for i in ti.grouped(x):
        v[i] += gravité * dt

    for i in ti.grouped(x):
        force = ti.Vector([0.0, 0.0, 0.0])
        for spring_offset in ti.static(indices_vois):
            j = i + spring_offset
            if 0 <= j[0] < n and 0 <= j[1] < n:
                x_ij = x[i] - x[j] #vecteur xij
                v_ij = v[i] - v[j] #vecteur vij
                d = x_ij.normalized()
                current_dist = x_ij.norm() #norme du vecteur xij
                original_dist = quad_size * float(i - j).norm()
                #2.force appliquée par les ressorts (allongement relatif)
                force += -ressort_Y * d * (current_dist / original_dist - 1)
                #2.amortissement du mvt relatif entre 2 masses
                force += -v_ij.dot(d) * d * amortissement_relatif * quad_size

        v[i] += force * dt

    for i in ti.grouped(x):
        #3.force d'amortissement visqueux
        v[i] *= ti.exp(-amortissement_visq * dt)
        #4.Collision avec la balle
        distance_au_centre = x[i] - centre[0]
        if distance_au_centre.norm() <= rayon:
            #projection de la vitesse
            normal = distance_au_centre.normalized()
            v[i] -= min(v[i].dot(normal), 0) * normal
        x[i] += dt * v[i]
##
```

```python
##
#mise à jour de la position des masses
#le terme i*n+j est du au fait qu'on accède à l'indice [i,j] sur un champ 1D
@ti.kernel
def maj_coord():
    for i, j in ti.ndrange(n, n):
        coord[i * n + j] = x[i, j]

#on définit l'espace ou sera visualisée la simulation
window = ti.ui.Window("Taichi Cloth Simulation on GGUI", (1024, 1024))
canvas = window.get_canvas()
canvas.set_background_color((1, 1, 1))
scene = ti.ui.Scene()
camera = ti.ui.Camera()

temps = 0.0
initialisation_masses()

while window.running:
    if temps > 1.5:
        # prochaine itération (boucle)
        initialisation_masses()
        temps = 0
#maj de l'état de système à chaque pas de temps
    for i in range(substeps):
        substep()
        temps+= dt
    maj_coord()

    camera.position(0.0, 0.0, 2.5)
    camera.lookat(0.0, 0.0, 0)
    scene.set_camera(camera)

    scene.point_light(pos=(0, 1, 2), color=(1, 1, 1))
    scene.ambient_light((0.5, 0.5, 0.5))
    scene.mesh(coord,
               indices=indices,
               per_vertex_color=couleurs,
               two_sided=True)

    # On dessine une balle plus petite pour eviter les penetrations visuelles
    scene.particles(centre, radius=rayon * 0.95, color=(0.8, 0.8, 0.8))
    canvas.scene(scene)
    window.show()
```