Moteur Physique pour les jeux vidéo

$5^{ème}$ GamiX

Chapitre 2 : Mouvements dans les mondes réel et virtuels

Partie 1 : Systèmes ponctuels

2025 – 2026

AHMED AMMAR

Compétences attendus

- Être capable de **simuler** un mouvement dans le monde virtuel.

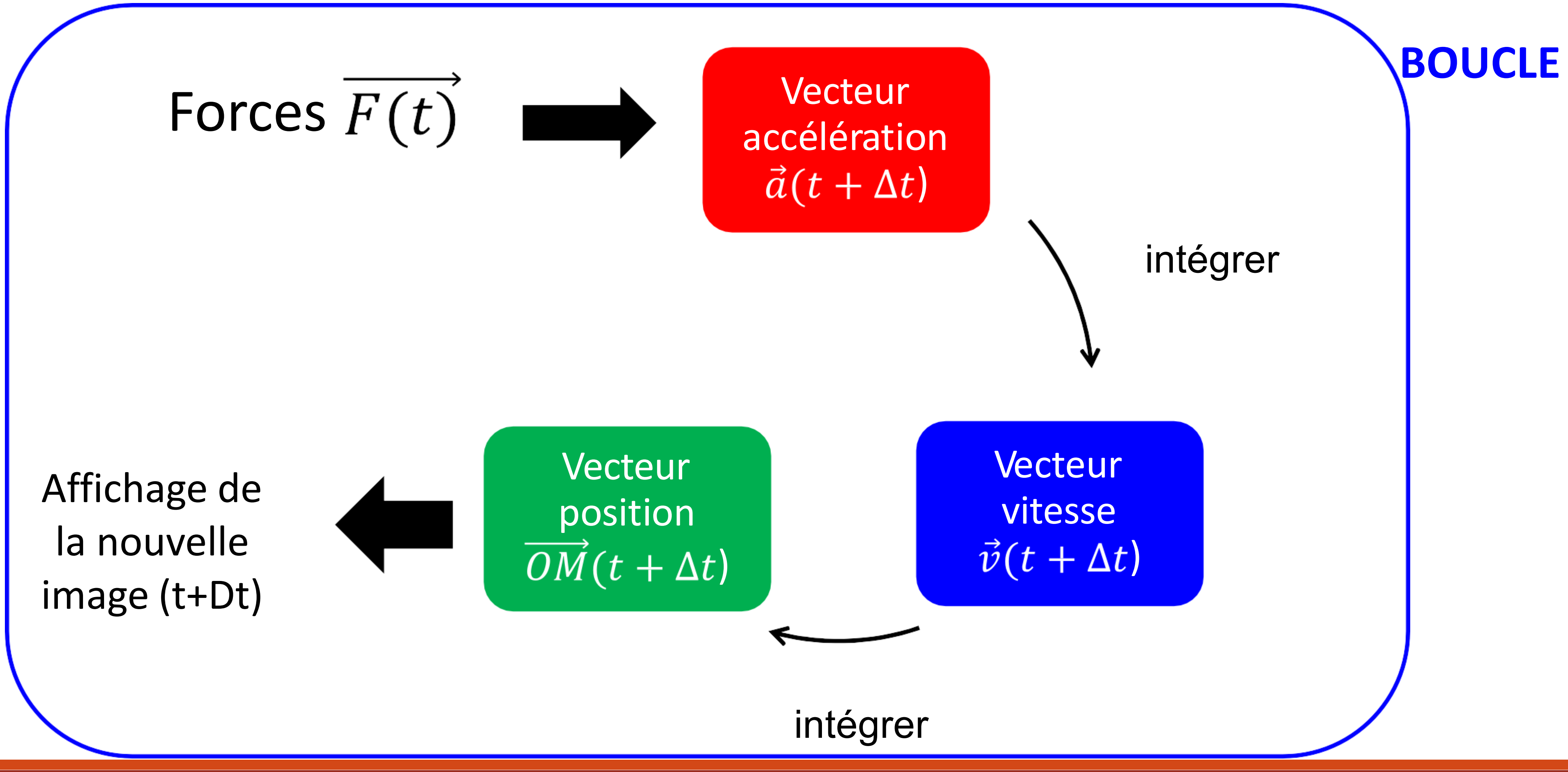
Prérequis

- Notions de base de cinématique du point matériel

- Notions de base de dynamique du point matériel.

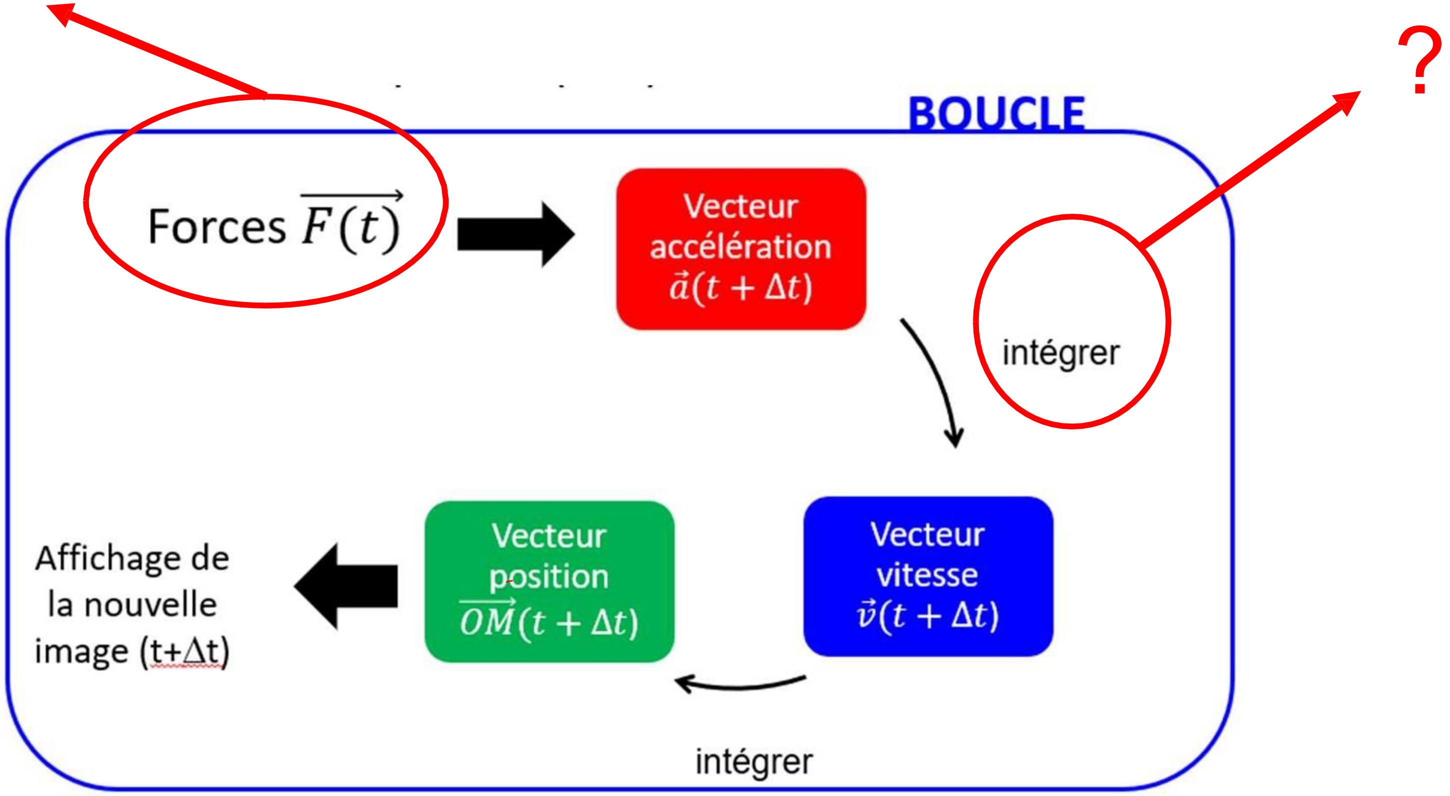
La programmation

Animation 3D temps réel = succession d’images (Frames) à une certaine fréquence (FPS)



La programmation : Les difficultés

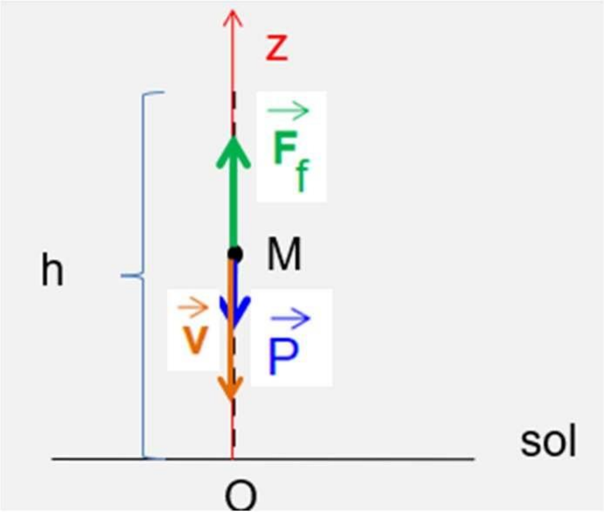
Les particules ne sont pas toujours toutes soumises aux mêmes forces.



Exemple: Etude de la chute libre en tenant compte de la présence de frottements fluides $\overrightarrow{F_f} = -\lambda\vec{v}$.

PFD ➡ $\forall t \in \mathbb{R}^+ \quad \ddot{z}(t) = -g - \dfrac{\lambda}{m}\dot{z}(t)$

Eq. Diff. du 2nd ordre !



$v = -\dot{z}$ ➡ $\forall t \in \mathbb{R}^+ \quad \dot{v}(t) = g - \dfrac{1}{\tau}v(t)$

$\tau = \dfrac{m}{\lambda}$ Temps caractéristique

Eq. Diff. du 1er ordre

Eq. Diff. du 1er ordre

Intégration numérique : Méthode d'Euler

On intègre sur $[0, t]$

$$\forall t \in \mathbb{R}^+ \qquad \dot{v}(t) = g - \frac{1}{\tau} v(t)$$

$$\forall t \in \mathbb{R}^+ \qquad \underbrace{\int_0^t \dot{v}(u)\,\mathrm{d}u}_{v(t)-v(0)} = \int_0^t \left(g - \frac{1}{\tau} v(u)\right) \mathrm{d}u$$

$$\forall t \in \mathbb{R}^+ \qquad v(t) - v(0) = g t - \frac{1}{\tau} \int_0^t v(u)\,\mathrm{d}u$$

La vitesse v peut varier fortement sur l'intervalle [0,t]

Intégration numérique : Méthode d'Euler

Si cet intervalle ($[0, t]$) est divisé en de nombreux sous-intervalles suffisamment petits pour que la vitesse y varie peu…?

On intègre sur $[t_i, t_{i+1}]$

$$\forall t \in \mathbb{R}^+ \qquad \dot{v}(t) = g - \frac{1}{\tau} v(t)$$

$$v(t_{i+1}) - v(t_i) = g \times \delta t - \frac{1}{\tau} \int_{t_i}^{t_{i+1}} v(u)\,\mathrm{d}u$$

$$\delta_t = t_{i+1} - t_i$$

Sur $[t_i, t_{i+1}]$ v varie peu, on peut l'approcher (l'approximer) par une constante et la sortir de l'intégrale !

Comment faire le choix de cette constante ?

Intégration numérique : Méthode d'Euler

Schéma d'Euler explicite

$$v(u) \approx v(t_i)$$

$$v(t_{i+1}) - v(t_i) \approx \left(g - \frac{v(t_i)}{\tau}\right) \times \delta t$$

$$\boxed{v_{i+1} = \left(1 - \frac{\delta t}{\tau}\right) v_i + g \times \delta t}$$

Schéma d'Euler implicite

$$v(u) \approx v(t_{i+1})$$

$$v(t_{i+1}) - v(t_i) \approx \left(g - \frac{v(t_{i+1})}{\tau}\right) \times \delta t$$

$$\boxed{v_{i+1} = \frac{1}{1 + \frac{\delta t}{\tau}} \left(v_i + g \times \delta t\right)}$$

Intégration numérique : Méthode d'Euler

- Permet de résoudre de façon approximative des équations différentielles ordinaires du premier ordre avec condition initiale.

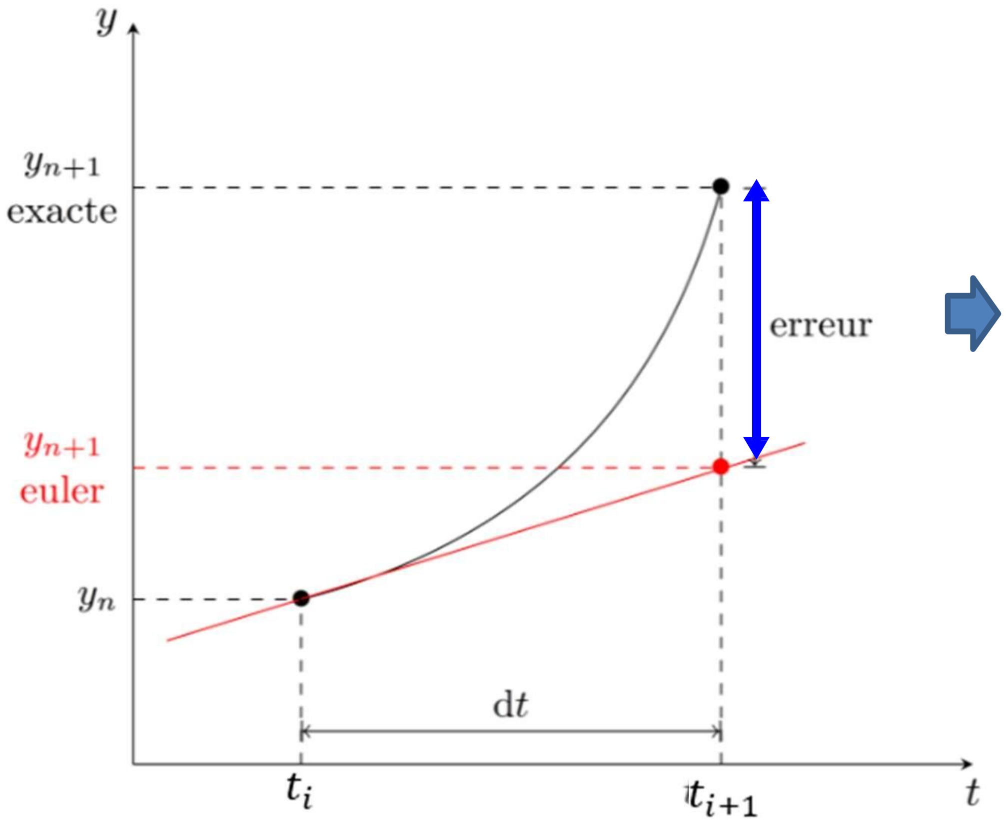
$$\begin{cases} \text{Condition initiale: } y(t=0) = y_0 \\ \text{Equation différentielle: } \dfrac{\mathrm{d}y}{\mathrm{d}t} = \boxed{f(t,y)} \end{cases}$$

$$v(t_{i+1}) - v(t_i) \approx \boxed{\left(g - \frac{v(t_i)}{\tau}\right)} \times \delta t$$

$$v(t_{i+1}) - v(t_i) \approx \boxed{\left(g - \frac{v(t_{i+1})}{\tau}\right)} \times \delta t$$

Intégration numérique : Méthode d'Euler

On évalue la valeur de la fonction y à l'instant t_{i+1} en calculant la pente de celle-ci ($\frac{dy}{dt}$) à l'instant t_i .



Dépend de
$\delta_{\mathrm{t}} = t_{\mathrm{i}+1} - t_{\mathrm{i}}$

Intégration numérique : Méthode d'Euler

Remarque:

D'une manière générale:

$$a(t) = \frac{F(t)}{m}$$

$$v(t + \Delta t) = v(t) + a(t)\Delta t$$

$$p_o(t + \Delta t) = p_o(t) + v(t)\Delta t$$

```
using UnityEngine;

public class FreeFall : MonoBehaviour
{
    public Vector3 velocity; // Vitesse du cube
    public float gravity = 9.81f; // Gravité
    public float dt = 0.002f; // Pas de temps
    public Vector3 position; // Position initiale du cube
    private CustomCube customCube; // Référence au script CustomCube

    void Start()
    {
        // Initialiser la position et la vitesse du cube
        velocity = Vector3.zero;
        position = Vector3.zero; // Le cube commence à l'origine

        // Obtenir la référence au script CustomCube
        customCube = GetComponent<CustomCube>();
    }

    // Utilisation de FixedUpdate pour des calculs physiques
    private void FixedUpdate()
    {
        // Appliquer la gravité (chute libre) via méthode d'Euler
        velocity += Vector3.down * gravity * dt;

        // Mise à jour de la position du cube
        position += velocity * dt;

        // Créer une matrice de translation manuellement
        Matrix4x4 translationMatrix = CreateTranslationMatrix(position);

        // Appliquer la matrice de transformation au cube
        customCube.ApplyTransformation(translationMatrix);
    }
```

```

    // Fonction pour créer une matrice de translation manuellement
    // Création de la matrice de translation (sans Matrix4x4.Translate)
    Matrix4x4 CreateTranslationMatrix(Vector3 translation)
    {
        Matrix4x4 matrix = Matrix4x4.identity; // Commence par une matrice identité

        // Remplir les valeurs de translation
        matrix.m03 = translation.x; // Déplacement en x
        matrix.m13 = translation.y; // Déplacement en y
        matrix.m23 = translation.z; // Déplacement en z

        return matrix; // Retourne la matrice de translation
    }
}
```

TP 1

Reprendre la simulation de la chute libre avec frottements visqueux en utilisant la méthode d'Euler.

La Méthode Runge-Kutta 4 (RK4)

$$\boldsymbol{y(t_{i+1}) = y(t_i) + \frac{h}{6}.(k_1 + 2.k_2 + 2.k_3 + k_4)} \qquad h = t_{i+1} - t_i$$

$$k_1 = f(\,y(t_i), t_i\,)$$

$$k_2 = f\left(\,y(t_i) + \frac{h}{2}k_1, t_i + \frac{h}{2}\right)$$

$$k_3 = f\left(\,y(t_i) + \frac{h}{2}k_2, t_i + \frac{h}{2}\right)$$

$$k_4 = f(\,y(t_i) + h.k_3, t_i + h)$$

<u>Exemple:</u>

$$\frac{dy}{dt} = f(y,t) \Rightarrow \begin{cases} \dfrac{dy}{dt} = -2.y.t \\ y(0) = 1 \end{cases}$$

TP2

Reprendre la simulation de la chute libre avec frottements visqueux en utilisant la méthode RK4.

La Méthode Runge-Kutta 4 (RK4)

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

Script Unity (1 référence de ressource) | 0 références
public class chutefrott : MonoBehaviour
{
    public Vector3 pos0;
    public Vector3 vel0;
    static Vector3 y0;

    private Vector3 pos;
    static Vector3 y;

    static Vector3 k1;
    static Vector3 k2;
    static Vector3 k3;
    static Vector3 k4;
    static  Vector3 g;
    static Vector3 vel;

    public Vector3 yp1;
    static float m=0.01f;
    static float lambda=0.01f;
    static float deltaT;
    static float tn;
```

```csharp
    // Start is called before the first frame update
    Message Unity | 0 références
    void Start()
    {
        tn = 0;



        g= new Vector3(0.0f, -9.81f, 0.0f);
        vel0= new Vector3(0.0f, 0.0f,0.0f);
        y0= new Vector3(0.0f, 0.0f,0.0f);

        pos0= new Vector3(0.0f, 300.0f,0.0f);
        pos= new Vector3(0.0f, 0.0f, 0.0f);
        deltaT = Time.fixedDeltaTime;

        vel= new Vector3(0.0f, 0.0f,0.0f);

    }
```

La Méthode Runge-Kutta 4 (RK4)

```
// Update is called once per frame
Message Unity | 0 références
void Update()
{
    tn += Time.fixedDeltaTime;


  vel= rk4(fff, vel,vel0);

  pos=rk4(ff2, vel, pos0);
  Debug.Log(pos);
    transform.position = pos;
}


1 référence
static Vector3 fff(Vector3 y, float t1, float deltaT1)
{
    Vector3 result;
    result= g-(lambda/m )*y ;
    return result;
  }

1 référence
static Vector3 ff2(Vector3 y2, float t2, float deltaT2)
{
   Vector3 result2;
   result2=-1.0f*y2;
   return result2;
}
```

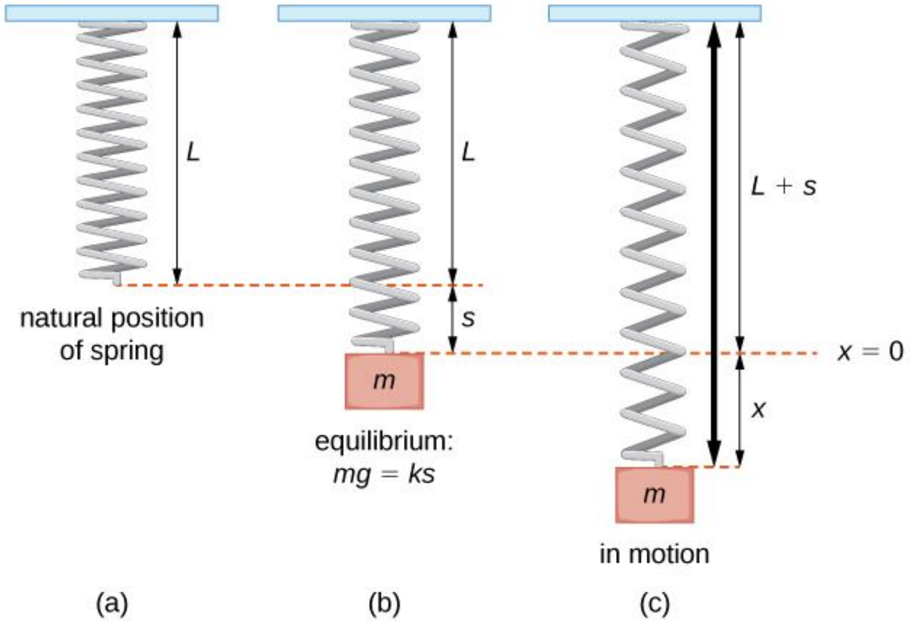
La Méthode Runge-Kutta 4 (RK4)

```
2 références
static Vector3 rk4 (Func<Vector3, float, float, Vector3> fff, Vector3 y, Vector3 y0)
{
    Vector3 yp1;


    k1= fff(y, tn, deltaT);
   // Debug.Log(y);
   k2 = fff(y + (deltaT / 2) * k1, tn,  tn + deltaT / 2);
    k3 = fff(y + (deltaT / 2) * k2,tn,  tn + deltaT / 2);
    k4 = fff(y + deltaT * k3, tn, tn + deltaT);

    yp1 =y+(deltaT / 6.0f) * (k1 + 2 * k2 + 2 * k3 + k4);

 return yp1;


}
```

Système Masse-Ressort-Amorti

Forces en Jeu

- ✓ **Force de rappel du ressort** ($\vec{F}_{\text{ressort}}$) :
  Selon la **loi de Hooke** :
  $$\vec{F}_{\text{ressort}} = -k\vec{x}$$
  où k est la constante de raideur du ressort, et $\vec{x}$ est le déplacement par rapport à la position d'équilibre.
- ✓ **Force d'amortissement** ($\vec{F}_{\text{amortissement}}$) :
  Proportionnelle à la vitesse $\vec{v}$ de la masse :
  $$\vec{F}_{\text{amortissement}} = -c\vec{v}$$
  où c est le coefficient d'amortissement.
- ✓ **Force de gravité** ($\vec{F}_{\text{gravité}}$) :
  La force de gravité agit verticalement :
  $$\vec{F}_{\text{gravité}} = m\vec{g}$$
  où g est l'accélération due à la gravité (généralement $g = 9.81$ m/s²).
- ✓ **Force d'inertie** ($\vec{F}_{\text{inertie}}$) :
  D'après la deuxième loi de Newton :
  $$\vec{F}_{\text{inertie}} = m\vec{a}$$
  où m est la masse et $\vec{a}$ l'accélération.



Système Masse-Ressort-Amorti

Équation de Mouvement

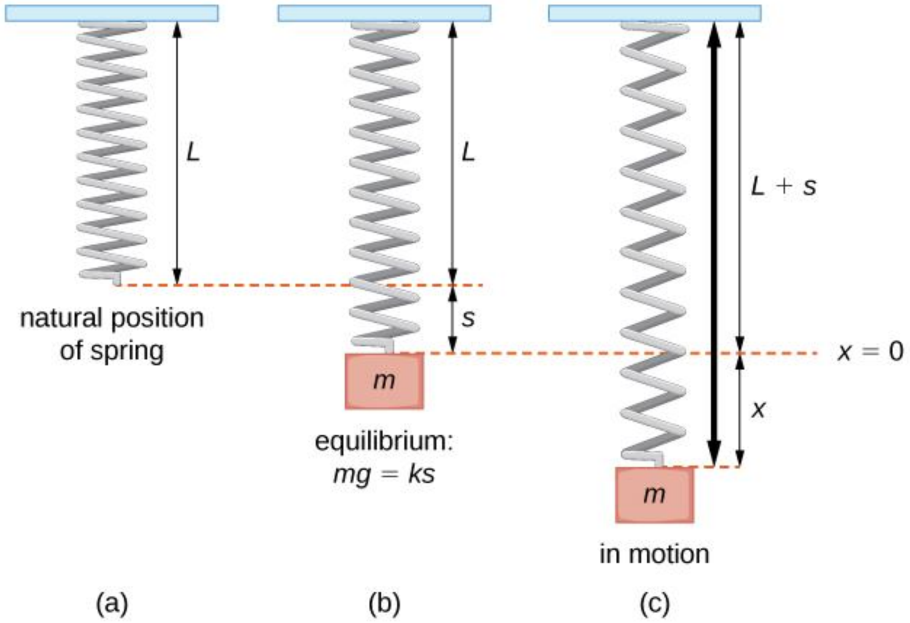
$\vec{F}_{\text{inertie}} = \vec{F}_{\text{ressort}} + \vec{F}_{\text{amortissement}} + \vec{F}_{\text{gravité}}$

En combinant ces forces, l'équation de mouvement du système est :

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = mg$$

Pour simplifier, on peut réécrire cette équation en termes de vitesse $v = \frac{dx}{dt}$ et d'accélération $a = \frac{d^2x}{dt^2}$:

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x - \frac{c}{m}\frac{dx}{dt} + g$$



Système Masse-Ressort-Amorti

Solution Numérique avec la Méthode de Runge-Kutta d'Ordre 4 (RK4)

Equation de mouvement du système

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x - \frac{c}{m}\frac{dx}{dt} + g$$

La méthode **RK4** est une méthode numérique pour résoudre les équations différentielles ordinaires. Voici comment l'appliquer à notre système :

1. **Définir les équations de premier ordre** :

   – Posons $v = \frac{dx}{dt}$ (vitesse).

   – L'équation de mouvement devient deux équations de premier ordre :

$$\begin{cases} \dfrac{dx}{dt} = v \\ \dfrac{dv}{dt} = -\dfrac{k}{m}x - \dfrac{c}{m}v + g \end{cases}$$

2. **Implémenter RK4** :

Pour chaque pas de temps h, nous calculons les valeurs intermédiaires :

$$k_1 = hf(t_n, x_n, v_n) = hv_n$$

$$l_1 = hg(t_n, x_n, v_n) = h\left(-\frac{k}{m}x_n - \frac{c}{m}v_n + g\right)$$

$$k_2 = hf\left(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}, v_n + \frac{l_1}{2}\right)$$

$$l_2 = hg\left(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}, v_n + \frac{l_1}{2}\right)$$

$$k_3 = hf\left(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}, v_n + \frac{l_2}{2}\right)$$

$$l_3 = hg\left(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}, v_n + \frac{l_2}{2}\right)$$

$$k_4 = hf(t_n + h, x_n + k_3, v_n + l_3)$$

$$l_4 = hg(t_n + h, x_n + k_3, v_n + l_3)$$

f et g sont définies comme :

$$\begin{cases} f(t, x, v) = v \\ g(t, x, v) = -\dfrac{k}{m}x - \dfrac{c}{m}v + g \end{cases}$$

3. **Mettre à jour les valeurs de x et v** :

   – Les nouvelles valeurs de x et v sont données par :

$$\boxed{x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)}$$

$$\boxed{v_{n+1} = v_n + \frac{1}{6}(l_1 + 2l_2 + 2l_3 + l_4)}$$

Avec ces formules, nous pouvons maintenant intégrer la position et la vitesse en tenant compte de la force de gravité dans le système masse-ressort-amortisseur.

Système Masse-Ressort-Amorti

Scripts c# : RK4Utility

```csharp
using UnityEngine;

public static class RK4Utility
{
    public static (Vector3, Vector3) RK4SolverMethod(Vector3 pos, Vector3 vel, Vector3 acc, float dt)
    {
        Vector3 k1 = dt * vel;
        Vector3 l1 = dt * acc;

        Vector3 k2 = dt * (vel + 0.5f * l1);
        Vector3 l2 = dt * (acc + 0.5f * l1);

        Vector3 k3 = dt * (vel + 0.5f * l2);
        Vector3 l3 = dt * (acc + 0.5f * l2);

        Vector3 k4 = dt * (vel + l3);
        Vector3 l4 = dt * (acc + l3);

        Vector3 newPos = pos + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0f;
        Vector3 newVel = vel + (l1 + 2 * l2 + 2 * l3 + l4) / 6.0f;

        return (newPos, newVel);
    }
}
```

Système Masse-Ressort-Amorti

Scripts c# : Simulation

```
Script Unity (1 référence de ressource) | 0 références
public class Simulation : MonoBehaviour
{
    public float damping = 0.1f; // Coefficient de frottement
    public float stiffness = 1.0f; // Constante de rappel du ressort
    public float mass = 1.0f; // Masse du cube
    public Vector3 initialPosition;
    public Vector3 initialVelocity;
    private Vector3 position;
    private Vector3 velocity;
    Message Unity | 0 références
    void Start()
    {
        position = initialPosition;
        velocity = initialVelocity;
    }
    Message Unity | 0 références
    void Update()
    {
        float dt = Time.deltaTime;
        Vector3 gravity = new Vector3(0, -9.81f, 0) * mass;
        Vector3 springForce = -stiffness * (position - initialPosition); // Force de rappel du ressort
        Vector3 dampingForce = -damping * velocity;
        Vector3 netForce = gravity + springForce + dampingForce;
        Vector3 acceleration = netForce / mass;

        // Appel de la méthode RK4SolverMethod de la classe utilitaire
        (position, velocity) = RK4Utility.RK4SolverMethod(position, velocity, acceleration, dt);

        transform.position = position;
    }
}
```

Système Ressort-Masse-Ressort-Masse-Amorti

Équations du Mouvement

Pour chaque masse, l'équation de mouvement peut être formulée comme suit :

1. **Masse 1 (m1)** :

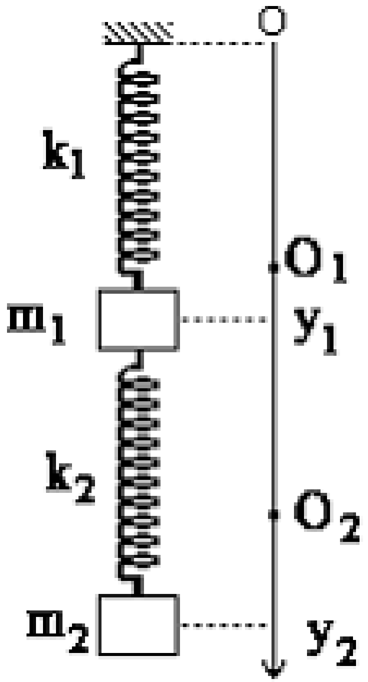
$$m_1 \frac{d^2 y_1}{dt^2} = -k_1(y_1 - L_0) - k_2(y_1 - y_2) - c_1 \frac{dy_1}{dt} + m_1 g$$

   - y_1 : position de la masse 1
   - k_1 : constante du ressort 1
   - L_0 : longueur au repos du ressort 1
   - k_2 : constante du ressort 2
   - y_2 : position de la masse 2
   - c_1 : coefficient d'amortissement 1
   - g : gravité

2. **Masse 2 (m2)** :

$$m_2 \frac{d^2 y_2}{dt^2} = -k_2(y_2 - y_1) - c_2 \frac{dy_2}{dt} + m_2 g$$

   - y_2 : position de la masse 2
   - c_2 : coefficient d'amortissement 2

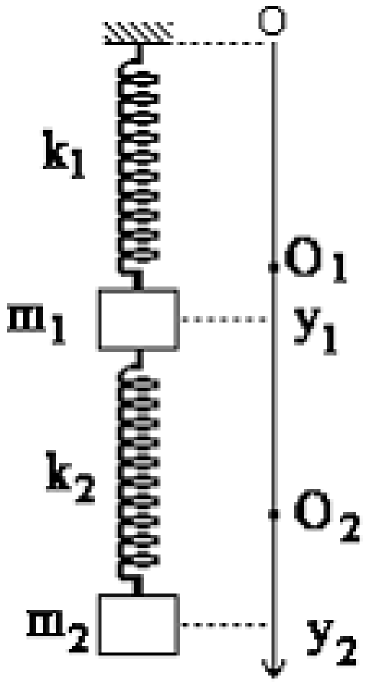


Système Ressort-Masse-Ressort-Masse-Amorti

Formulation pour RK4

Pour appliquer la méthode RK4, nous devons reformuler les équations ci-dessus en un système d'équations de premier ordre. Nous définissons les vitesses et les accélérations comme suit :

- $v_1 = \frac{dy_1}{dt}$
- $v_2 = \frac{dy_2}{dt}$



Les équations du premier ordre deviennent :

- $\frac{dy_1}{dt} = v_1$
- $\frac{dv_1}{dt} = \frac{-k_1(y_1 - L_0) - k_2(y_1 - y_2) - c_1 v_1 + m_1 g}{m_1}$
- $\frac{dy_2}{dt} = v_2$
- $\frac{dv_2}{dt} = \frac{-k_2(y_2 - y_1) - c_2 v_2 + m_2 g}{m_2}$

Système Ressort-Masse-Ressort-Masse-Amorti

Formulation pour RK4

1. **Définir les fonctions** :
   - Soit $f(y_1, v_1, y_2, v_2)$ une fonction qui représente les dérivées des positions et vitesses :

$$f(y_1, v_1, y_2, v_2) = \begin{pmatrix} v_1 \\ \dfrac{-k_1(y_1 - L_0) - k_2(y_1 - y_2) - c_1 v_1 + m_1 g}{m_1} \\ v_2 \\ \dfrac{-k_2(y_2 - y_1) - c_2 v_2 + m_2 g}{m_2} \end{pmatrix}$$

2. **Utiliser cette fonction dans la méthode RK4** :
   - La méthode RK4 peut alors être réécrite en utilisant f comme suit :
   - **Calcul des dérivées initiales** :

$$k_1 = f(y_1, v_1, y_2, v_2)$$

   - **Calcul des dérivées intermédiaires** :

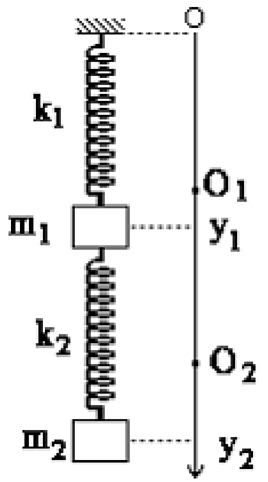
$$k_2 = f\left(y_1 + \frac{h}{2}k_{1y1}, v_1 + \frac{h}{2}k_{1v1}, y_2 + \frac{h}{2}k_{1y2}, v_2 + \frac{h}{2}k_{1v2}\right)$$

$$k_3 = f\left(y_1 + \frac{h}{2}k_{2y1}, v_1 + \frac{h}{2}k_{2v1}, y_2 + \frac{h}{2}k_{2y2}, v_2 + \frac{h}{2}k_{2v2}\right)$$

$$k_4 = f\left(y_1 + hk_{3y1}, v_1 + hk_{3v1}, y_2 + hk_{3y2}, v_2 + hk_{3v2}\right)$$

3. **Mise à jour des états** :
   - Ensuite, les mises à jour pour les positions et vitesses peuvent être calculées de la manière suivante :

$$\boxed{\begin{aligned} y_1^{n+1} &= y_1^n + \frac{h}{6}(k_{1y1} + 2k_{2y1} + 2k_{3y1} + k_{4y1}) \\ v_1^{n+1} &= v_1^n + \frac{h}{6}(k_{1v1} + 2k_{2v1} + 2k_{3v1} + k_{4v1}) \\ y_2^{n+1} &= y_2^n + \frac{h}{6}(k_{1y2} + 2k_{2y2} + 2k_{3y2} + k_{4y2}) \\ v_2^{n+1} &= v_2^n + \frac{h}{6}(k_{1v2} + 2k_{2v2} + 2k_{3v2} + k_{4v2}) \end{aligned}}$$



Système Ressort-Masse-Ressort-Masse-Amorti

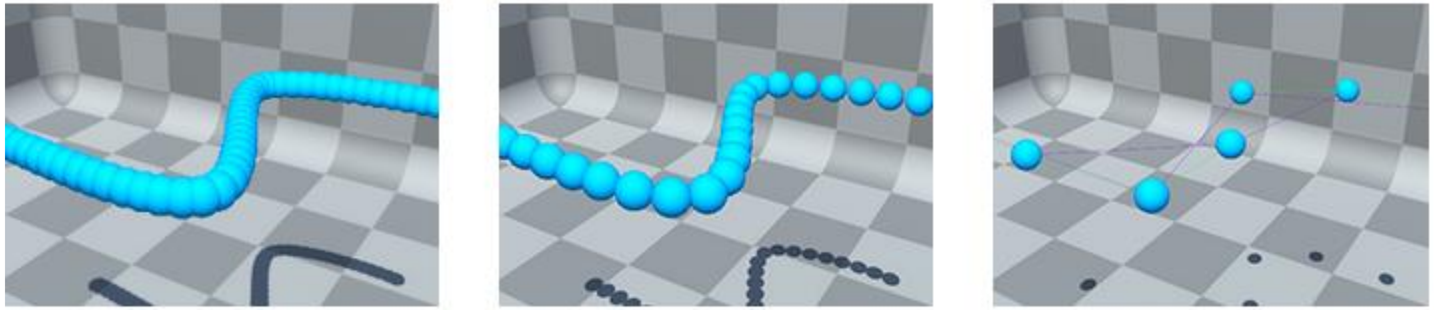
```
Message Unity | 0 références
void Update()
{
    // Mise à jour de la simulation
    RK4Step();
    UpdateMassPositions();
}
```

```
4 références
Vector4 Derivative(Vector4 state, Vector3 force1, Vector3 force2)
{
    float y1 = state.x;
    float v1 = state.y;
    float y2 = state.z;
    float v2 = state.w;

    float dy1 = v1;
    float dv1 = (-k1 * (y1 - L0) - k2 * (y1 - y2) - c1 * v1) / m1 + 9.81f;
    float dy2 = v2;
    float dv2 = (-k2 * (y2 - y1) - c2 * v2) / m2 + 9.81f;

    return new Vector4(dy1, dv1, dy2, dv2);
}

1 référence
void UpdateMassPositions()
{
    mass1.transform.position = pos1;
    mass2.transform.position = pos2;
}
}
```

Système Ressort-Masse-Ressort-Masse-Amorti

```
void RK4Step()
{
    // Forces initiales
    Vector3 force1 = Vector3.zero;
    Vector3 force2 = Vector3.zero;
    float g = 9.81f; // Accélération due à la gravité

    // Calculer les forces sur chaque masse
    force1 += -k1 * (pos1.y - L0) * Vector3.up; // Force du ressort 1
    force1 += -k2 * (pos1.y - pos2.y) * Vector3.up; // Force du ressort 2
    force1 += -c1 * vel1; // Force d'amortissement 1
    force1 += m1 * g * Vector3.down; // Force gravitationnelle (vers le bas)

    force2 += -k2 * (pos2.y - pos1.y) * Vector3.up; // Force du ressort 2
    force2 += -c2 * vel2; // Force d'amortissement 2
    force2 += m2 * g * Vector3.down; // Force gravitationnelle (vers le bas)

    // Système d'équations
    Vector4 state = new Vector4(pos1.y, vel1.y, pos2.y, vel2.y);

    // Définir les fonctions dérivées
    Vector4 k1_local = dt * Derivative(state, force1, force2); // Corrigé : renommé en k1_local
    Vector4 k2_local = dt * Derivative(state + 0.5f * k1_local, force1, force2); // Corrigé : renommé en k2_local
    Vector4 k3 = dt * Derivative(state + 0.5f * k2_local, force1, force2);
    Vector4 k4 = dt * Derivative(state + k3, force1, force2);

    // Mettre à jour l'état
    pos1.y += (k1_local.x + 2 * k2_local.x + 2 * k3.x + k4.x) / 6.0f;
    vel1.y += (k1_local.y + 2 * k2_local.y + 2 * k3.y + k4.y) / 6.0f;
    pos2.y += (k1_local.z + 2 * k2_local.z + 2 * k3.z + k4.z) / 6.0f;
    vel2.y += (k1_local.w + 2 * k2_local.w + 2 * k3.w + k4.w) / 6.0f;
}
```

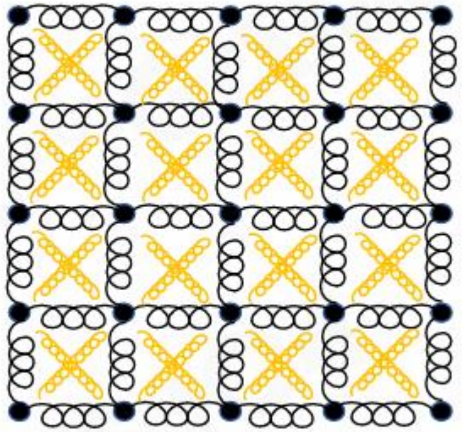
Systèmes à plusieurs particules



Resolution: 1   Resolution: 0.5   Resolution: 0.1

- Spring force
  - $\vec{F}^{s} = k_{\text{spring}} \Delta \vec{x}$
- Spring damping force
  - $\vec{F}^{D} = -k_{\text{damping}} \vec{v}$

Sont utilisés pour simuler des systèmes complexes.



Systèmes à plusieurs particules

A un instant donné:

Force exercée sur une particule $f = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$

$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ Position de la particule

$\Rightarrow \ddot{x} = \frac{f}{m}$

2nd ordre !

$$\left.\begin{aligned} \ddot{x} &= \frac{f}{m} \\ \dot{x} &= v \end{aligned}\right\} \quad \begin{aligned} \dot{v} &= \frac{f}{m} \\ \dot{x} &= v \end{aligned}$$

$\Rightarrow$ On définit notre **vecteur d'état** $\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix}$

- Si les masses des particules sont différentes.
- Si les forces exercées sur les particules sont différentes.

$$\Rightarrow \begin{pmatrix} x \\ v \\ f \\ m \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ v_1 \\ v_2 \\ v_3 \\ f_1 \\ f_2 \\ f_3 \\ m \end{pmatrix}$$

Systèmes à plusieurs particules

$$\frac{d}{dt}\begin{pmatrix} x \\ v \\ f \\ m \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \frac{f_1}{m} \\ \frac{f_2}{m} \\ \frac{f_3}{m} \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \Rightarrow \quad \frac{dy}{dt} = f(y, t)$$

Masses et forces sont supposées constantes

Systèmes à plusieurs particules

<u>Remarque:</u>

- Un système à N particules différentes est décrit par un vecteur à 10*N composantes.

$$\left[\overbrace{\begin{pmatrix} x \\ v \\ f \\ m \end{pmatrix} \dots\dots\dots\dots\dots\dots \begin{pmatrix} x \\ v \\ f \\ m \end{pmatrix}}^{N \text{ fois}}\right]$$

Systèmes à plusieurs particules

- Pour une bonne implémentation, il est intéressant de pouvoir modifier , voir étendre, le nombre et la nature des forces sans changer la configuration du système de particules.