Physics 203 - Livre du cours

Table des matières

Session 1 : Géométrie et Calcul Vectoriel

Définitions

Un vecteur est une grandeur ayant une direction et une magnitude.

On dit qu'il est de dimension n si il a n composantes.

Dans le plan, $n = 2$, donc un vecteur a 2 composantes, 2 dimensions.

Représentation

Un vecteur est représenté par une flèche. Il peut être placé n'importe où dans le plan. On le place habituellement par rapport au contexte de ce qu'il représente.

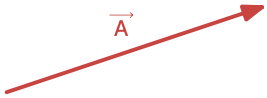


Fig. 1. – Un vecteur

Notation vectorielle

Même si le vecteur est défini par ses magnitude et direction, on utilise une notation vectorielle, qui décompose le vecteur selon les vecteurs $\hat{\imath}$ et $\hat{\jmath}$ du plan cartésien. On retrouve ses 2 dimensions représentées en colonne.

Exemple.

$$\vec{v} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} = 2\hat{\imath} + 3\hat{\jmath}$$

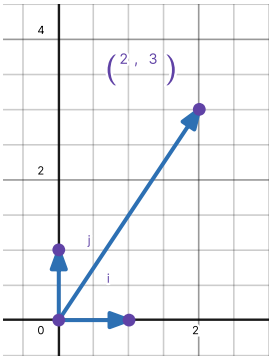


Fig. 2. – Le vecteur $\vec{v}$

Déplacement

Un déplacement peut se décrire par le vecteur reliant le point A (départ) au point B (arrivée) :

$$\overrightarrow{AB} = \begin{pmatrix} x_B - x_A \\ y_B - y_A \end{pmatrix}$$

Des vecteurs célèbres en physique

- **Le vecteur position**

Souvent noté $\vec{r}$ (radius), représente la position d'un point dans l'espace en partant de l'origine.

- **Le vecteur vitesse**

Noté $\vec{v}$, représente la vitesse d'un objet, souvant représenté partant du centre de l'objet.

- **Le vecteur accélération**

Noté $\vec{a}$, représente l'accélération d'un objet, souvant représenté partant du centre de l'objet.

Magnitude et direction

La longueur du vecteur, aussi appelée magnitude, est notée $\|\vec{v}\|$, et parfois simplement $|\vec{v}|$.

En 2D :

On utilise le théorème de Pythagore pour calculer la norme (autre nom de la magnitude) :

$$\|\vec{v}\| = \sqrt{v_x^2 + v_y^2}$$

Vecteur Unitaire ($\hat{u}$) :

Vecteur de longueur 1 direction $\vec{v}$:

$$\hat{u} = \frac{\vec{v}}{\|\vec{v}\|}$$

Un vecteur unitaire est un vecteur de longueur 1, il est souvent utilisé pour représenter une direction sans avoir à se soucier de la magnitude.

Opérations de Base

Soit $\vec{u} = \begin{pmatrix} u_x \\ u_y \end{pmatrix}$ et $\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$.

Addition :

$$\vec{u} + \vec{v} = \begin{pmatrix} u_x + v_x \\ u_y + v_y \end{pmatrix}$$

Soustraction :

$$\vec{u} - \vec{v} = \begin{pmatrix} u_x - v_x \\ u_y - v_y \end{pmatrix}$$

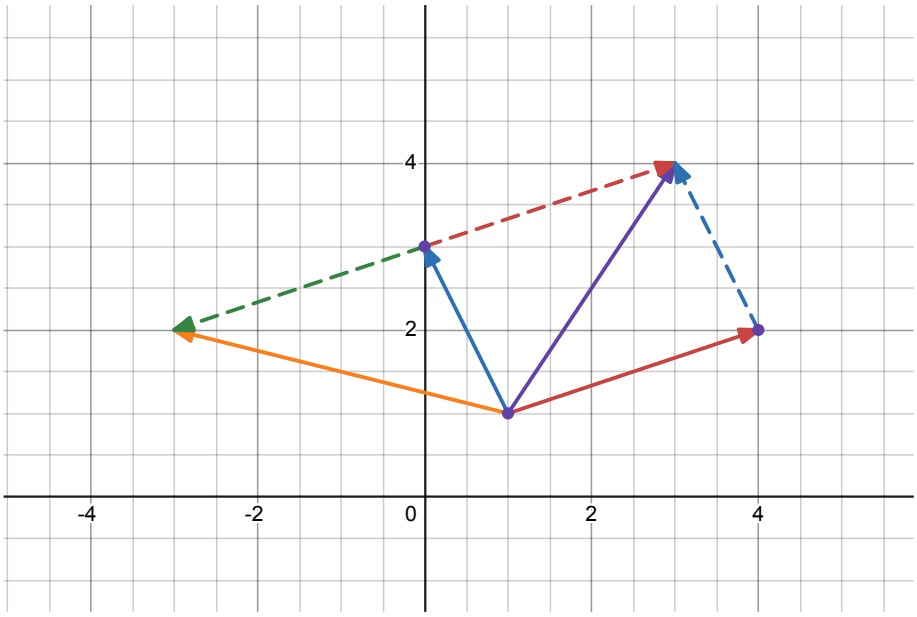


Fig. 3. – Le vecteur $\vec{u}$ (rouge) et le vecteur $\vec{v}$ (bleu)

Multiplication par scalaire (k) :

$$k \cdot \vec{u} = \begin{pmatrix} ku_x \\ ku_y \end{pmatrix}$$

Produit Scalaire

Résultat : un **nombre** (scalaire).

Formule algébrique :

$$\vec{u} \cdot \vec{v} = u_x v_x + u_y v_y + u_z v_z$$

Formule géométrique :

$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\theta)$$

1. Si $\vec{u} \cdot \vec{v} = 0$, alors $\vec{u} \perp \vec{v}$.
2. Si $\vec{u} \cdot \vec{v} > 0$, alors l'angle entre les vecteurs est inférieur à 90°. (devant)
3. Si $\vec{u} \cdot \vec{v} < 0$, alors l'angle entre les vecteurs est supérieur à 90°. (derrière)

Angle entre 2 vecteurs

On isole $\cos(\theta)$. :

$$\cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \cdot \|\vec{v}\|}$$

$$\theta = \arccos\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \cdot \|\vec{v}\|}\right)$$

Projections

Projection de $\vec{u}$ sur $\vec{v}$. On utilise le produit scalaire, dans la direction sur laquelle on projette.

$$\text{proj}_v(\vec{u}) = \left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{v}\|^2}\right) \cdot \vec{v}$$

Et si $\vec{v}$ est normalisé (direction de la droite sur laquelle on projète), on peut simplifier :

$$\text{proj}_v(\vec{u}) = (\vec{u} \cdot \vec{v}) \cdot \vec{v}$$

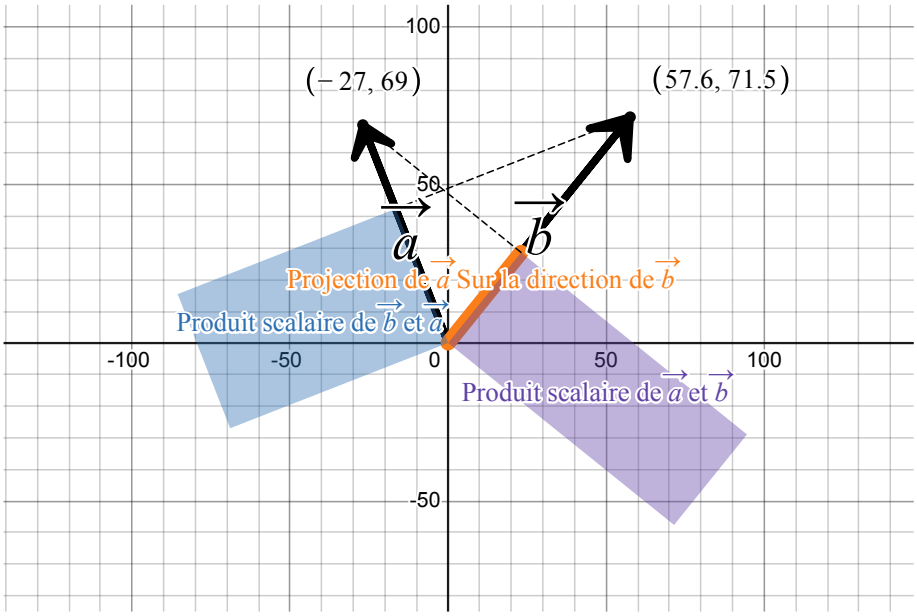


Fig. 4. – Produit scalaire, projection

Produit Vectoriel (3D)

Résultat : un **vecteur** perpendiculaire.

$$\vec{w} = \vec{a} \times \vec{b}$$

Commutativité :

Attention, le produit vectoriel n'est pas commutatif ! On a un changement de signe :

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$$

Magnitude :

$$\|\vec{a} \times \vec{b}\| = \|\vec{a}\| \cdot \|\vec{b}\| \cdot \sin(\theta)$$

Direction :

Le produit vectoriel est perpendiculaire aux deux vecteurs :

$$a \times b \perp a$$

$$a \times b \perp b$$

On utilise la rêgle de la main droite pour trouver la direction du produit vectoriel.
- Le pouce pointe sur a
- l'index sur b
- le majeur sur $a \times b$.

Calcul (Déterminant) :

Le déterminant est un outil algébrique qui permet de trouver les solutions d'un problème de n équations avec n inconnues. Cette solution représente, en quelque sorte une direction émergente (perpendiculaire) des équations.

$$a = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \text{ et } b = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

Session 2 : Cinématique 2D et Mouvement de Projectile

Résumé de la Session :

- **Introduction à la cinématique :**
  - Branche de la mécanique qui décrit le mouvement des objets sans considérer les causes du mouvement (les forces).
  - Contrairement à la dynamique, qui relie le mouvement aux forces.
  - Notre objectif dans ce bloc est de développer les outils mathématiques pour décrire précisément *comment* les objets se déplacent.

La position, la vitesse et l'accélération sont des concepts qui permettent de décrire précisément comment un objet se comporte.

La relation entre ces concepts est la suivante :

- La position est le vecteur qui indique la localisation d'un point dans l'espace.
- La vitesse est le vecteur qui indique la vitesse du point, il est défini comme la variation (dérivée) de la position par rapport au temps.
- L'accélération est le vecteur qui indique l'accélération du point, il est défini comme la variation (dérivée) de la vitesse par rapport au temps.

Lorsque la position est constante, le point est au repos, immobile. Sa vitesse et son accélération sont nulles.

Lorsque la vitesse est constante, le point se déplace à vitesse constante en ligne droite. Son accélération est nulle.

Lorsque l'accélération est constante, la vitesse augmente constamment, le point se déplace de plus en plus vite.

1. Introduction et Rappels

- **Brève révision des vecteurs :**
  - Rappel de la définition d'un vecteur comme une quantité possédant une magnitude (longueur, norme) et une direction.
  - Représentation d'un vecteur en 2D à l'aide de ses composantes :

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

.

  - Opérations vectorielles de base (addition et soustraction) :

$$\vec{a} + \vec{b} = \begin{pmatrix} a_x + b_x \\ a_y + b_y \end{pmatrix}$$

$$\vec{a} - \vec{b} = \begin{pmatrix} a_x - b_x \\ a_y - b_y \end{pmatrix}$$

- Multiplication d'un vecteur par un scalaire :

$$k\vec{a} = \begin{pmatrix} ka_x \\ ka_y \end{pmatrix}$$

.

2. Position en 2D

- **Vecteur Position :**
  - Pour décrire la localisation d'un point (ou d'un objet considéré comme un point) dans un plan bidimensionnel, nous utilisons le **vecteur position**, noté $\vec{r}$ (ou parfois $\vec{s}$ ou $\vec{x}$).
  - Si nous définissons un système de coordonnées cartésiennes avec un axe horizontal (x) et un axe vertical (y), le vecteur position d'un point P de coordonnées (x, y) est donné par :

$$\vec{r} = \begin{pmatrix} x \\ y \end{pmatrix} = x\hat{\imath} + y\hat{\jmath}$$

    où $\hat{\imath} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ est le vecteur unitaire dans la direction de l'axe x, et $\hat{\jmath} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ est le vecteur unitaire dans la direction de l'axe y.
  - Le vecteur position pointe de l'origine du système de coordonnées vers la position de l'objet.
  - La **magnitude** du vecteur position, $|\vec{r}| = \sqrt{x^2 + y^2}$, représente la distance de l'objet à l'origine.
  - La **direction** du vecteur position peut être donnée par l'angle θ qu'il forme avec l'axe x, où $\tan(\theta) = \frac{y}{x}$.

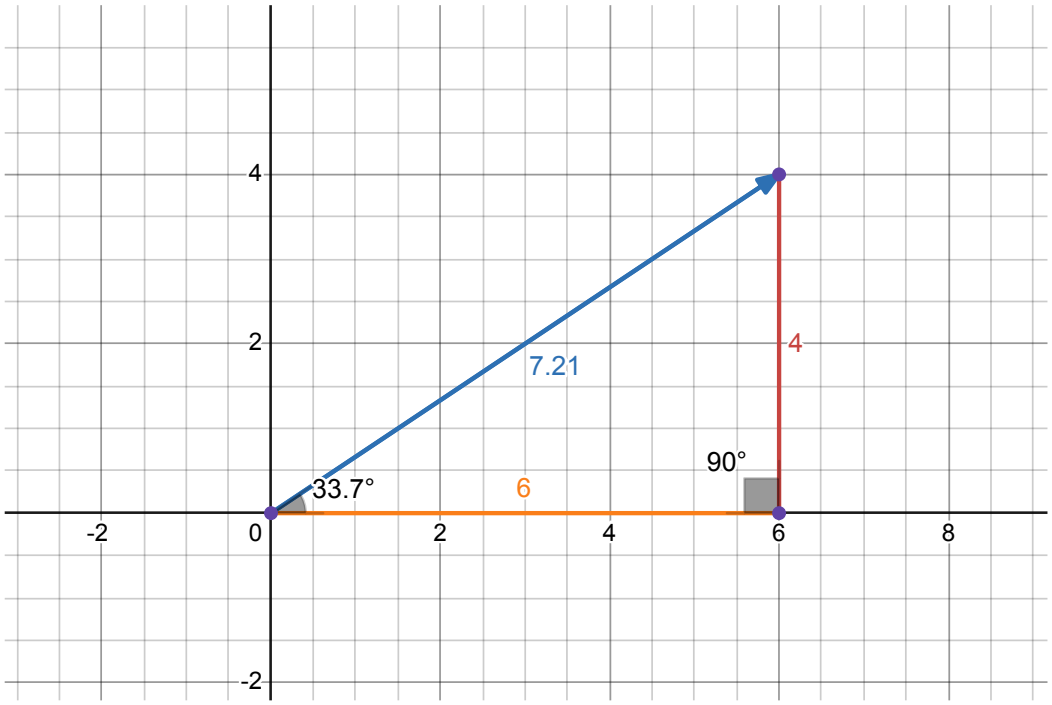


Fig. 5. – Plan 2D avec vecteurs position et vitesse

- **Trajectoire :**

- Si la position d'un objet change au cours du temps, nous pouvons décrire son mouvement en spécifiant son vecteur position en fonction du temps :

$$\vec{r}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

.

- L'ensemble des points atteints par l'objet au cours de son mouvement forme sa **trajectoire**. La trajectoire est une courbe dans l'espace (ici, en 2D).
- Exemples de trajectoires :
  - **Mouvement rectiligne uniforme :**

    $\vec{r}(t) = \begin{pmatrix} x_0 + v_x t \\ y_0 + v_y t \end{pmatrix}$,

    où x_0, y_0, v_x, v_y sont des constantes. La trajectoire est une ligne droite.
  - **Mouvement circulaire uniforme :**

    $\vec{r}(t) = \begin{pmatrix} R\cos(\omega t) \\ R\sin(\omega t) \end{pmatrix}$, où R est le rayon et ω la vitesse angulaire. La trajectoire est un cercle.
  - **Mouvement parabolique (projectile) :**

    $\vec{r}(t) = \begin{pmatrix} v_{0x} t \\ y_0 + v_{0y} t - \frac{1}{2} g t^2 \end{pmatrix}$ (sous l'effet de la gravité). La trajectoire est une parabole.
- Visualisation de différentes trajectoires et des vecteurs position correspondants à différents instants.

- **Démo cinématique :** Pour visualiser les concepts de cinématique, consultez la démo : file:../scenes/session02.html

3. Vitesse en 2D

- **Vitesse Moyenne :**

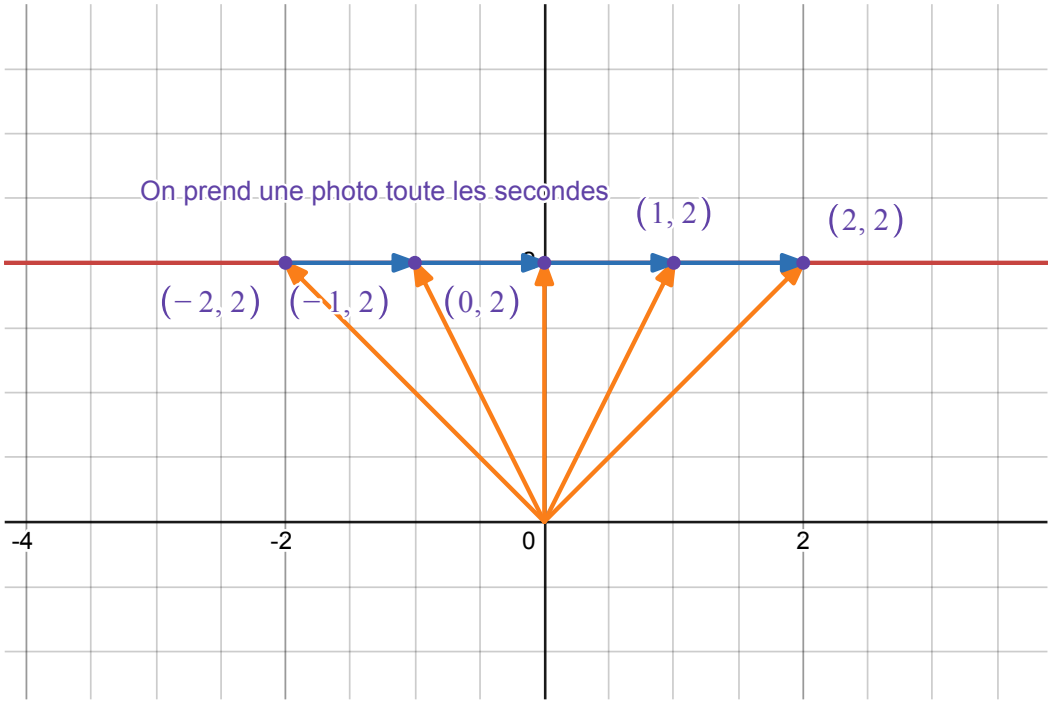


Fig. 6. – Plan 2D avec vecteurs position et vitesse

- Considérons un objet qui se déplace de la position $\vec{r}_i$ à l'instant t_i à la position $\vec{r}_f$ à l'instant t_f.
- Le **déplacement** de l'objet pendant cet intervalle de temps $\Delta t = t_f - t_i$ est le vecteur :

$$\Delta\vec{r} = \vec{r}_f - \vec{r}_i = \begin{pmatrix} x_f - x_i \\ y_f - y_i \end{pmatrix} = \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

- Le **vecteur vitesse moyenne** $\vec{v}_m$ est défini comme le rapport du déplacement au temps écoulé :

$$\vec{v}_{\text{moy}} = \frac{\Delta\vec{r}}{\Delta t} = \frac{\vec{r}_f - \vec{r}_i}{t_f - t_i} = \begin{pmatrix} \frac{\Delta x}{\Delta t} \\ \frac{\Delta y}{\Delta t} \end{pmatrix} = \begin{pmatrix} v_{\text{moy},x} \\ v_{\text{moy},y} \end{pmatrix}$$

- La vitesse moyenne est un vecteur dont la direction est la même que celle du déplacement, et dont la magnitude est le déplacement total divisé par le temps écoulé.

Exemple : Si une voiture se déplace de la position $\vec{r}_i = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ à la position $\vec{r}_f = \begin{pmatrix} 10 \\ 5 \end{pmatrix}$ au cours d'un intervalle de temps $\Delta t = 5$ secondes, alors le déplacement de la voiture est $\Delta\vec{r} = \begin{pmatrix} 10 \\ 5 \end{pmatrix}$ et la vitesse moyenne est $\vec{v}_{\text{moy}} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ m/s.

- **Vitesse Instantanée :**
  - Pour décrire la vitesse de l'objet à un instant précis t, nous utilisons la notion de **vitesse instantanée**, $\vec{v}(t)$.
  - Mathématiquement, la vitesse instantanée est définie comme la limite de la vitesse moyenne lorsque l'intervalle de temps Δt tend vers zéro :

$$\vec{v}(t) = \lim_{\Delta t \to 0} \frac{\Delta\vec{r}}{\Delta t} = \frac{d\vec{r}}{dt}$$

  - En termes de composantes, la vitesse instantanée est la dérivée des composantes de la position par rapport au temps :

$$\vec{v}(t) = \begin{pmatrix} \frac{dx(t)}{dt} \\ \frac{dy(t)}{dt} \end{pmatrix} = \begin{pmatrix} v_x(t) \\ v_y(t) \end{pmatrix}$$

où $v_{x(t)}$ est la composante de la vitesse selon l'axe x, et $v_{y(t)}$ est la composante de la vitesse selon l'axe y à l'instant t.

- La **magnitude** de la vitesse instantanée, $|\vec{v}(t)| = \sqrt{v_{x(t)}^2 + v_{y(t)}^2}$, est appelée **vitesse scalaire**.
- La **direction** de la vitesse instantanée est tangente à la trajectoire de l'objet au point considéré. Visualisation de ce concept avec des exemples de trajectoires courbes.

Application aux différents types de trajectoire :

1. Mouvement rectiligne uniforme :

- Pour $\vec{r}(t) = \begin{pmatrix} x_0 + v_x t \\ y_0 + v_y t \end{pmatrix}$, la vitesse instantanée est :

$$\vec{v}(t) = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

- Le vecteur vitesse est **constant** : direction, magnitude et composantes ne changent pas au cours du temps. Par exemple, un objet avec $\vec{r}(0) = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ et $\vec{v} = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$ se déplace horizontalement à 3 m/s.

2. Mouvement circulaire uniforme :

- Pour $\vec{r}(t) = \begin{pmatrix} R\cos(\omega t) \\ R\sin(\omega t) \end{pmatrix}$, la vitesse instantanée est :

$$\vec{v}(t) = \begin{pmatrix} -R\omega \sin(\omega t) \\ R\omega \cos(\omega t) \end{pmatrix} = \begin{pmatrix} -\omega y(t) \\ \omega x(t) \end{pmatrix}$$

- La magnitude est constante : $|\vec{v}| = R\omega$. À l'instant $t = \frac{\pi}{2\omega}$ (quand l'objet est en haut du cercle), $\vec{r} = \begin{pmatrix} 0 \\ R \end{pmatrix}$ et $\vec{v} = \begin{pmatrix} -R\omega \\ 0 \end{pmatrix}$: la vitesse est purement horizontale et tangente au cercle.

3. Mouvement parabolique (projectile) :

- Pour $\vec{r}(t) = \begin{pmatrix} v_{0x} t \\ y_0 + v_{0y} t - \frac{1}{2} g t^2 \end{pmatrix}$, la vitesse instantanée est :

$$\vec{v}(t) = \begin{pmatrix} v_{0x} \\ v_{0y} - gt \end{pmatrix}$$

- La composante horizontale v_x reste constante, tandis que la composante verticale v_y diminue linéairement. Par exemple, avec $v_{0x} = 10$ m/s, $v_{0y} = 20$ m/s et $g = 9.81$ m/s², à $t = 2$ s : $\vec{v} = \begin{pmatrix} 10 \\ 20 - 9.81 * 2 \end{pmatrix} = \begin{pmatrix} 10 \\ 0.38 \end{pmatrix}$ m/s. Le projectile est quasiment à son apogée.

4. Accélération en 2D

- **Accélération Moyenne :**
  - Si la vitesse d'un objet change au cours du temps, l'objet est en train d'accélérer.
  - Le **vecteur accélération moyenne** $\vec{a}_{\text{moy}}$ pendant un intervalle de temps $\Delta t = t_f - t_i$ est défini comme le rapport du changement de vitesse au temps écoulé :

$$\vec{a}_{\text{moy}} = \frac{\Delta \vec{v}}{\Delta t} = \frac{\vec{v}_f - \vec{v}_i}{t_f - t_i} = \begin{pmatrix} \frac{\Delta v_x}{\Delta t} \\ \frac{\Delta v_y}{\Delta t} \end{pmatrix} = \begin{pmatrix} a_{\text{moy},x} \\ a_{\text{moy},y} \end{pmatrix}$$

  - L'accélération moyenne est un vecteur dont la direction est celle du changement de vitesse.

- **Accélération Instantanée :**
  - L'**accélération instantanée** $\vec{a}(t)$ décrit la manière dont la vitesse d'un objet change à un instant précis t. Elle est définie comme la dérivée de la vitesse par rapport au temps :

$$\vec{a}(t) = \lim_{\Delta t \to 0} \frac{\Delta \vec{v}}{\Delta t} = \frac{d\vec{v}}{dt}$$

  - En termes de composantes, l'accélération instantanée est la dérivée des composantes de la vitesse par rapport au temps, ou la deuxième dérivée des composantes de la position par rapport au temps :

$$\vec{a}(t) = \begin{pmatrix} a_x(t) \\ a_y(t) \end{pmatrix} = \begin{pmatrix} \frac{dv_x(t)}{dt} \\ \frac{dv_y(t)}{dt} \end{pmatrix} = \begin{pmatrix} \frac{d^2x(t)}{dt^2} \\ \frac{d^2y(t)}{dt^2} \end{pmatrix}$$

  - L'accélération peut changer la magnitude de la vitesse (l'objet accélère ou décélère), sa direction, ou les deux en même temps.

  Cas particulier : Accélération constante. Si l'accélération $\vec{a}$ est constante, alors $\vec{a}(t) = \vec{a} = \begin{pmatrix} a_x \\ a_y \end{pmatrix}$, où a_x et a_y sont des constantes. Dans ce cas, nous pouvons **intégrer** les équations de l'accélération pour obtenir la vitesse et la position en fonction du temps :

$$\vec{v}(t) = \vec{v}_0 + \vec{a}t = \begin{pmatrix} v_{0x} + a_x t \\ v_{0y} + a_y t \end{pmatrix}$$

$$\vec{r}(t) = \int_{t_0}^{t} \vec{v}(t)dt = \vec{r}_0 + \vec{v}_0 t + \frac{1}{2}\vec{a}t^2 = \begin{pmatrix} x_0 + v_{0x}t + \frac{1}{2}a_x t^2 \\ y_0 + v_{0y}t + \frac{1}{2}a_y t^2 \end{pmatrix}$$

  où $\vec{v}_0 = \begin{pmatrix} v_{0x} \\ v_{0y} \end{pmatrix}$ est la vitesse initiale et $\vec{r}_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ est la position initiale.

  C'est ce cas particulier qui sera crucial pour l'étude du mouvement de projectile sous l'effet de la gravité.

Exemples complet par type de trajectoire :

Nous pouvons maintenant reprendre nos 3 exemples et finaliser notre étude de leurs mouvements.

1. Mouvement rectiligne uniforme:

Position :

$$\vec{r}(t) = \begin{pmatrix} x_0 + v_x t \\ y_0 + v_y t \end{pmatrix}$$

où x_0 et y_0 sont les positions initiales, et v_x et v_y sont les vitesses initiales.

Vitesse (dérivée de la position) :

$$\vec{v}(t) = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

Accélération (dérivée de la vitesse) :

$$\vec{a}(t) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

La dérivée d'un vecteur constant est nulle. Aucune accélération n'est nécessaire pour maintenir le mouvement rectiligne uniforme.

2. Mouvement circulaire uniforme :

Position :

- À l'instant t, le vecteur position est :

$$\vec{r}(t) = \begin{pmatrix} R\cos(\omega t) \\ R\sin(\omega t) \end{pmatrix}$$

Vitesse (dérivée de la position) :

- Dérivons chaque composante par rapport au temps :

$$\vec{v}(t) = \begin{pmatrix} \frac{d(R\cos(\omega t))}{dt} \\ \frac{d(R\sin(\omega t))}{dt} \end{pmatrix} = \begin{pmatrix} -R\omega\sin(\omega t) \\ R\omega\cos(\omega t) \end{pmatrix}$$

Accélération (dérivée de la vitesse) :

- Dérivons à nouveau chaque composante :

$$\vec{a}(t) = \begin{pmatrix} -\frac{d(R\omega\sin(\omega t))}{dt} \\ \frac{d(R\omega\cos(\omega t))}{dt} \end{pmatrix} = \begin{pmatrix} -R\omega^2\cos(\omega t) \\ -R\omega^2\sin(\omega t) \end{pmatrix}$$

Relation avec la position :

- On observe que :

$$\vec{a}(t) = -\omega^2 \begin{pmatrix} R\cos(\omega t) \\ R\sin(\omega t) \end{pmatrix} = -\omega^2\vec{r}(t)$$

L'accélération est constante et bien dirigée vers le centre du cercle.

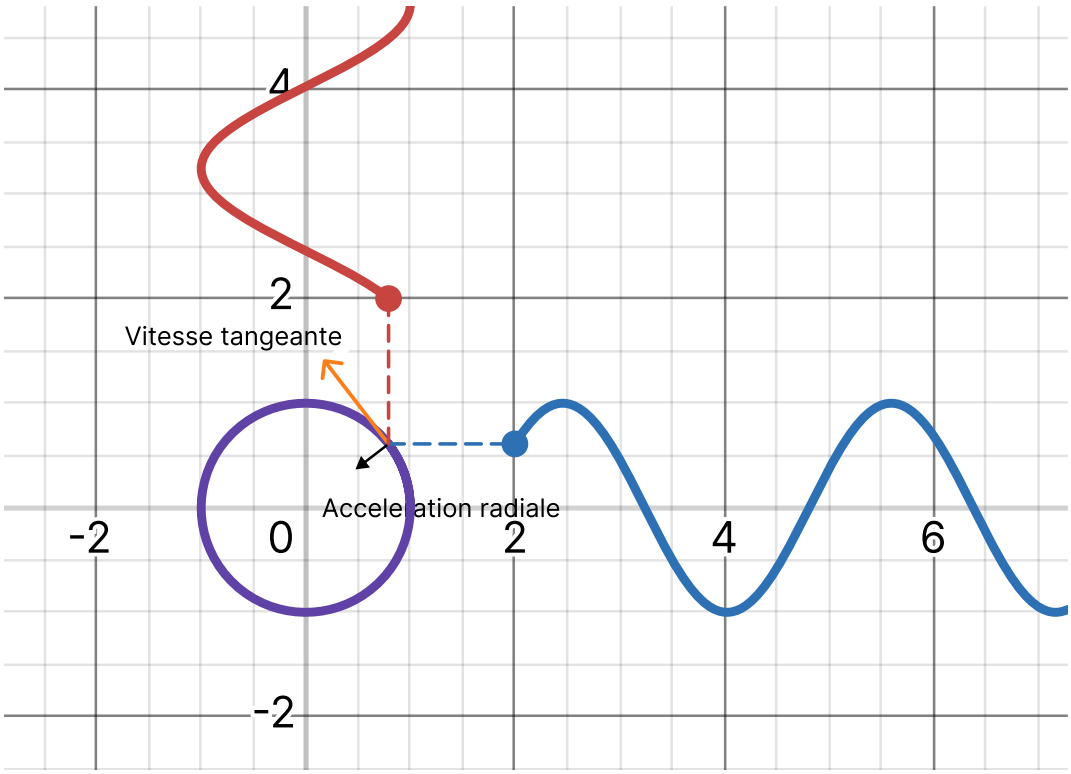


Fig. 7. – Le mouvement circulaire uniforme

3. Mouvement parabolique (projectile) :

- L'équation de la trajectoire est :

$$\vec{r}(t) = \begin{pmatrix} v_{0x}t \\ y_0 + v_{0y}t - \frac{1}{2}gt^2 \end{pmatrix}$$

  - Pour $\vec{v}(t) = \begin{pmatrix} v_{0x} \\ v_{0y} - gt \end{pmatrix}$, l'accélération instantanée est :

$$\vec{a}(t) = \begin{pmatrix} 0 \\ -g \end{pmatrix}$$

  - L'accélération est **constante** et dirigée verticalement vers le bas.

Avec $g = 9.81$ m/s², quel que soit l'instant t, $\vec{a} = \begin{pmatrix} 0 \\ -9.81 \end{pmatrix}$ m/s². Cette accélération change uniquement la composante verticale de la vitesse. La composante horizontale reste inchangée.

Session 3 : Introduction des lois de Newton et la méthode Euler

Objectifs de la Session :

- Réviser les sessions précédentes (la cinématique).
- Introduire les lois de Newton.
- Relier les précédentes sessions (la cinématique) aux lois de Newton.
- Introduire la méthode Euler pour résoudre les équations de la cinématique.
- Setup de git, vs code, live preview et Tinymist, pour avoir un acces en ligne et en local à l'ensemble des documents.
- Réaliser un premier travail pratique: Observer une de résolution d'équations cinématiques avec la méthode Euler dans un exemple simple.

Bloc 1 : Révision des sessions précédentes

Voir le document « session_02.typ » pour une révision des sessions précédentes.

Bloc 2 : Introduction des lois de Newton

Voir le document « newton.typ » pour une introduction aux lois de Newton.

Bloc 2 : Introduction de la méthode Euler

Voir le document « euler.typ » pour une introduction à la méthode Euler.

Les Forces de Frottement

Introduction

Dans le vide spatial (et dans un moteur physique naïf), un objet lancé garde sa vitesse pour l'éternité (1ère loi de Newton).

Sur Terre, tout finit par s'arrêter. Pourquoi ? À cause des interactions avec l'environnement. Ces interactions sont modélisées par des Forces de Frottement.

Il existe deux grandes familles à maîtriser pour le jeu vidéo :
1. Le Frottement Sec (Solide contre Solide).
2. Le Frottement Visqueux (Solide dans un Fluide).

1. Le Frottement Sec (Modèle de Coulomb)

C'est le frottement de contact : un pneu sur la route, une caisse sur le sol, ou une boule de billard qui glisse. Tous les frottements qui « accrochent ».

Il ne dépend pas de la vitesse (tant que ça bouge), ni de la surface (l'aire) de contact (contrairement à l'intuition).

Il dépend de deux choses :
1. **La rugosité des matériaux** (Coefficient μ).
2. **Le poids apparent** (Force Normale N).

$$\|\vec{f}\| = \mu \cdot \|\vec{N}\|$$

Direction : Toujours opposée à la vitesse (ou à la force qui tente de bouger l'objet). Il compense la force qui tente de bouger l'objet, et le garde immobile. Il va s'adapter exactement pour annuler votre poussée, jusqu'à un seuil critique.

La Force Normale (N)

C'est la force avec laquelle les deux surfaces sont pressées l'une contre l'autre.

- Sur un sol plat horizontal : $N = mg$ (Le Poids).
- Sur une pente : $N = mg\cos(\theta)$ (La composante perpendiculaire au sol).
- Dans un virage relevé (F1) : N augmente à cause de la force centrifuge.

-Bien assis sur son siège : $N = mg\cos(\alpha)$ (la composante perpendiculaire au sol).

Distinction : Statique vs Cinétique

Avez-vous remarqué qu'il est plus dur de **commencer** à pousser une armoire lourde que de la **maintenir** en mouvement ? On pousse de plus en plus fort et tout d'un coup on passe le seuil critique, et l'armoire commence à bouger beaucoup plus facilement.

A. Frottement Statique (μ_s)
- L'objet est immobile ($v = 0$).
- La force est « intelligente » : elle s'adapte exactement pour annuler votre poussée, jusqu'à un seuil critique.
- **Seuil de rupture :** $F_{\max} = \mu_s N$.

B. Frottement Cinétique (μ_k)
- L'objet glisse ($v > 0$).
- La force est **constante**, peu importe la vitesse.
- $F_k = \mu_k N$.

En général : $\mu_s > \mu_k$. (C'est pour ça que l'armoire « décolle » d'un coup).

2. Le Frottement Visqueux (Fluide)

C'est la résistance de l'air (Drag) ou de l'eau. Contrairement au frottement sec, celui-ci dépend énormément de la vitesse, ainsi que la forme de l'objet.

Cas 1 : Vitesse faible (Loi de Stokes) Pour les particules dans le miel ou la poussière.

$$\vec{f} = -k \cdot \vec{v}$$

(Proportionnel à la vitesse).

Cas 2 : Vitesse élevée (Loi quadratique) Pour les voitures, les avions, les balles.

$$\vec{f} = -C \cdot \|v\|^2 \cdot \hat{v}$$

(Proportionnel au **carré** de la vitesse).

Conséquence : La Vitesse Terminale Si un objet tombe, la gravité (mg) est constante, mais le frottement fluide augmente. À un moment, les deux s'annulent : l'objet ne peut plus accélérer.

Exemple. **Comparaison pour le Gameplay :**

- **Frottement Sec (Billard, Glisse)** : L'objet ralentit linéairement (v diminue de manière constante). Il s'arrête net.
- **Frottement Fluide (Amortissement)** : L'objet ralentit de moins en moins vite. Il ne s'arrête théoriquement jamais complètement (approche asymptotique de 0).

TP : Le Laboratoire de Chocs 1D

Objectifs du TP

Nous allons implémenter la réponse physique d'une collision entre deux boules de masses différentes.

Pour simplifier le problème et isoler les mathématiques de l'impulsion, nous travaillons dans un « Laboratoire 1D » :

- Pas de gravité.
- Mouvement uniquement sur l'axe X.
- Pas de friction pour l'instant.

1. Analyse de la Scène

Le code de démarrage (`Template.js`) contient :

- Une **Boule Rouge (A)** : Lourde ($m = 5$), lancée vers la droite.
- Une **Boule Verte (B)** : Légère ($m = 1$), immobile (actuellement désactivée).
- Une interface **GUI** : Pour modifier les masses, la vitesse et le temps en direct.

État actuel : La boule rouge traverse le vide et rebondit bêtement sur les murs (inversion simple de vitesse).

2. Rappel Théorique : L'Impulsion

Lors d'un choc, nous devons calculer une impulsion j (scalaire) et l'appliquer aux deux objets.

Formule de l'Impulsion (avec Masse Réduite) :

$$j = -(1 + e) \cdot v_{\text{rel}} \cdot \left(\frac{m_A m_B}{m_A + m_B} \right)$$

- e : Coefficient de restitution (1 = Élastique, 0 = Mou).
- v_{rel} : Vitesse relative de rapprochement

$$(\vec{v}_A - \vec{v}_B) \cdot \vec{n}$$

.

Application (Action-Réaction) :

$$\vec{v}'_A = \vec{v}_A + \left(\frac{j}{m_A} \right) \vec{n}$$

$$\vec{v}'_B = \vec{v}_B - \left(\frac{j}{m_B}\right)\vec{n}$$

3. Vos Missions (Code)

Étape A : Activer la Boule B Dans le fichier `init()`, décommentez les lignes créant la deuxième boule. Relancez : les boules devraient se traverser comme des fantômes.

Étape B : Implémenter le « Solver » (Fonction `resolveBallCollision`) Repérez la zone `TODO ÉTUDIANT`. Vous devez traduire la physique en JavaScript (Three.js).

1. Calculez la Normale $\vec{n}$ (Vecteur unitaire de B vers A).
2. Calculez la Vitesse Relative v_{rel} (Produit scalaire).
3. **Sécurité :** Si $v_{\text{rel}} > 0$ (elles s'éloignent), on arrête (`return`).
4. Calculez la Masse Réduite et l'Impulsion j.
5. Appliquez le changement aux vitesses `velA` et `velB`.

Indice : Utilisez `addScaledVector(vector, scale)`.

4. Expérimentations (Observations)

Une fois votre code fonctionnel, utilisez la GUI pour tester ces scénarios physiques classiques.

Scénario 1 : Le Pendule de Newton
- Réglez $m_A = 5$ et $m_B = 5$ (Masses égales).
- Restitution $e = 1$.
- **Observation :** La boule rouge doit s'arrêter net. La verte doit partir avec toute la vitesse.

Scénario 2 : Le Mur
- Réglez $m_A = 1$ (Légère) et $m_B = 100$ (Très lourde).
- **Observation :** La rouge rebondit en arrière. La verte bouge à peine.

Scénario 3 : Le Bulldozer (Effet Catapulte)
- Réglez $m_A = 100$ (Très lourde) et $m_B = 1$ (Légère).
- **Observation :** La rouge continue presque sans ralentir. À quelle vitesse part la verte ? (Indice : $2 \times v_A$).

Scénario 4 : La Pâte à modeler
- Mettez la restitution $e = 0$.
- **Observation :** Les boules doivent se coller et avancer ensemble à une vitesse moyenne.

Astuce Debug

Si vos boules disparaissent ou accélèrent à l'infini :

1. Vérifiez le signe dans l'application de l'impulsion ($-j$ pour B).
2. Vérifiez la condition `if (vRel > 0) return;`.
3. Utilisez le slider « Time Scale » pour mettre le temps au ralenti (0.1) et voir exactement ce qui se passe au moment de l'impact.

Cahier d'Exercices : Vecteurs & Physique

Thème 1 : Vecteurs et Produits Scalaires

Ces exercices visent à maîtriser les outils mathématiques de base pour la détection et l'IA.

Exercice 1 : Le Garde et le Voleur.

Contexte : Dans un jeu d'infiltration, un garde regarde dans une direction donnée par le vecteur directeur $\vec{D}$. Un voleur se trouve à une position P dans la pièce. Le garde se trouve à la position G.

On considère que le garde a un champ de vision de 180° (il voit tout ce qui est devant lui).

Données :
- Position du Garde : $G(10, 10)$
- Direction du regard : $\vec{D} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ (Regarde vers l'Est)
- Position du Voleur : $P(5, 12)$

Question : En utilisant le produit scalaire, déterminer si le voleur est « Devant » ou « Derrière » le garde.

Exercice 2 : La Racing Line (Projections).

Contexte : Pour aider une IA à conduire, on définit la trajectoire idéale comme une ligne droite entre deux points A et B. La voiture se trouve actuellement au point V.

On souhaite connaitre la distance latérale entre la voiture et cette trajectoire idéale (l'écart de conduite).

Données :
- Départ de la ligne : $A(0, 0)$
- Fin de la ligne : $B(100, 0)$
- Position Voiture : $V(50, 5)$

Question :
1. Calculer le vecteur $\overrightarrow{AV}$.
2. Projeter ce vecteur sur la direction de la piste $\overrightarrow{AB}$.
3. En déduire la distance de la voiture à la ligne.

Exercice 3 : Le Radar de Vitesse (Vitesse Relative).

Contexte : Un radar mesure la vitesse de rapprochement sur sa ligne de visée. Pour savoir ce que le radar affiche, il faut projeter les vitesses des véhicules sur l'axe qui les relie.

Données (Positions et Vitesses) :

- Police (A) :
  - Position : $P_A = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$
  - Vitesse : $\vec{v}_A = \begin{pmatrix} 0 \\ 100 \end{pmatrix}$ (Roule vers le Nord à 100 km/h)
- Cible (B) :
  - Position : $P_B = \begin{pmatrix} 0 \\ 500 \end{pmatrix}$ (Est située 500m au Nord de la police)
  - Vitesse : $\vec{v}_B = \begin{pmatrix} 85 \\ 85 \end{pmatrix}$ (Roule vers le Nord-Est à ≈ 120 km/h)

Questions :
1. Calculer le vecteur direction de visée $\vec{D} = P_B - P_A$.
2. En déduire la normale de visée $\vec{n}$ (le vecteur unitaire).
3. Projeter les vitesses $\vec{v}_A$ et $\vec{v}_B$ sur $\vec{n}$ (Produit Scalaire).
4. La police est-elle en train de rattraper la cible ? (Est-ce que v_A projeté $> v_B$ projeté ?)

Thème 2 : Forces, Gravité et Frottements

Application des lois de Newton pour simuler des mouvements réalistes.

Exercice 4 : Le Drag Race (Code).

Contexte : Une voiture accélère en ligne droite.

- Force Moteur : $F_{\text{moteur}} = 5000\text{N}$ (Constante).
- Friction de l'air : $F_{\text{air}} = -k \cdot v^2$ (Opposée au mouvement).
- Masse : $m = 1000$ kg, Coefficient $k = 0.8$.

Objectif Code : Implémenter ce système dans Three.js. Afficher la vitesse en temps réel. Observer la « Vitesse Terminale » : le moment où la voiture n'accélère plus car la friction compense exactement le moteur.

Exercice 5 : Le Sniper (Balistique). **Contexte :** On tire un projectile soumis uniquement à la gravité $\vec{g} = \begin{pmatrix} 0 \\ -9.81 \end{pmatrix}$.

Données :
- Vitesse initiale : $v_0 = 50$ m/s.
- Angle de tir : $\theta = 45°$.
- Hauteur initiale : $y_0 = 0$.

Objectif Code : Trouver ce résultat dans une simulation avec la méthode d'Euler.

Exercice 6 : Le Cube sur la Pente (Code). **Contexte :** Reprise de l'exercice théorique sur le plan incliné.

Mise en place : Créer un plan (sol) dans Three.js dont on peut modifier l'inclinaison (rotation X) via une interface (GUI). Poser un cube dessus.

Logique à coder :
- Calculer la force de gravité parallèle à la pente : $P_x = mg\sin(\theta)$.
- Calculer la friction statique max : $f_{s,\max} = \mu_s mg\cos(\theta)$.
- **Condition :** Tant que $P_x < f_{s,\max}$, la vitesse reste à 0. Sinon, le cube accélère.

Test : Vérifier si le cube se met à glisser exactement à l'angle prévu (ex: 30° pour $\mu_s = 0.57$).

Exercice 7 : L'Ascenseur Spatial (Ressorts). **Contexte :** Une boule est suspendue dans le vide, attachée à un point fixe par un ressort invisible.

Loi de Hooke : $\vec{F} = -k \cdot (L - L_0) \cdot \vec{n}$ (Où L est la longueur actuelle et L_0 la longueur au repos).

Objectif Code : Simuler ce ressort. Ajouter une force d'amortissement (Damping) $F = -b \cdot v$ pour que la boule finisse par se stabiliser et ne rebondisse pas à l'infini.

Définition 1 (Thème 3 : Collisions et Impulsions)

Gestion des chocs et conservation de la quantité de mouvement.

Exercice 8 : Le Bumper Car (Code). **Contexte :** Arène fermée avec 5 boules de masses identiques.

Objectif Code : Mettre en place la boucle de détection « Narrow Phase » : vérifier chaque paire de boules (i, j). Si `distance < radius * 2`, résoudre la collision élastique ($e = 1$).

Observer le chaos et vérifier que les boules ne « fusionnent » pas (stabilité).

Exercice 9 : Le Matériau Mystère. **Contexte :** On lâche 3 boules de la même hauteur $H = 10m$. Elles tombent sur le sol (masse infinie).

- Boule A : Remonte à 10m.
- Boule B : Remonte à 5m.
- Boule C : Reste collée au sol.

Question : Déterminer le coefficient de restitution e pour chaque boule. Rappel : $v_{\text{après}} = -e \cdot v_{\text{avant}}$.

Exercice 10 : Le Rocket Jump. **Contexte :** Une explosion se produit à la position $E(0, 0, 0)$. Un joueur se trouve à la position $P(2, 0, 0)$.

On veut propulser le joueur avec une impulsion radiale.

Règle :
- Direction : Du centre de l'explosion vers le joueur.
- Intensité : $J = \frac{100}{\text{distance}^2}$.

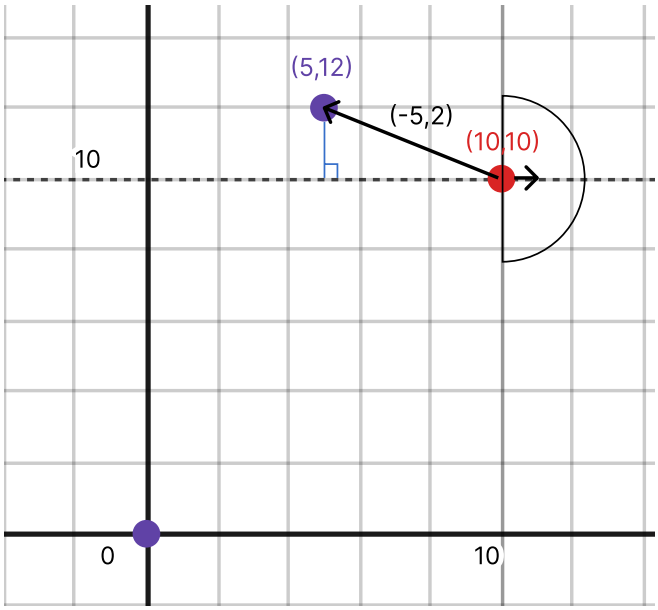
Question : Calculer le vecteur Impulsion $\vec{J}$ à appliquer au joueur.

Corrigé : Vecteurs & Physique

Thème 1 : Vecteurs et Produits Scalaires

Solutions détaillées.

Solution 1 : Le Garde et le Voleur



1. **Calcul du vecteur Garde → Voleur ($\vec{V}$) :**

$$\vec{V} = P - G = \begin{pmatrix} 5 - 10 \\ 12 - 10 \end{pmatrix} = \begin{pmatrix} -5 \\ 2 \end{pmatrix}$$

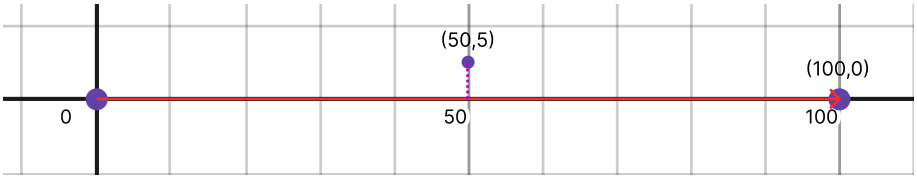
2. **Produit Scalaire (Test de vision)** : On projette la position du voleur sur la direction du regard.

$$\text{Dot} = \vec{V} \cdot \vec{D} = (-5 \times 1) + (2 \times 0) = -5$$

3. **Conclusion** : Le résultat est négatif ($-5 < 0$). L'angle entre le regard et le voleur est donc supérieur à 90°.

   Réponse : Le voleur est Derrière le garde.

Solution 2 : La Racing Line



1. **Vecteur Voiture ($\overrightarrow{AV}$) :**

$$\overrightarrow{AV} = V - A = \begin{pmatrix} 50 \\ 5 \end{pmatrix}$$

2. **Direction de la piste ($\vec{u}$)** : Le vecteur $\overrightarrow{AB} = \begin{pmatrix} 100 \\ 0 \end{pmatrix}$. Son vecteur unitaire (normalisé) est $\vec{u} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.
3. **Projection (Distance parcourue sur la piste)** :

$$\text{Proj} = \overrightarrow{AV} \cdot \vec{u} = (50 \times 1) + (5 \times 0) = 50$$

   Le point projeté sur la ligne est donc $H(50, 0)$.
4. **Distance latérale (Écart)** : C'est la distance entre $V(50, 5)$ et $H(50, 0)$.

   Réponse : La voiture est à 5 mètres de la ligne idéale.

Solution 3 : Le Radar (Vitesse Relative)

1. **Vecteur de visée $\vec{D}$** :

$$\vec{D} = P_B - P_A = \begin{pmatrix} 0 \\ 500 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 500 \end{pmatrix}$$

2. **Normale de visée $\vec{n}$** : On normalise $\vec{D}$. Comme il est purement vertical :

$$\vec{n} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

3. **Projections des vitesses (Produit Scalaire)** :
   - Police : $v_{A_{\text{proj}}} = \begin{pmatrix} 0 \\ 100 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 100 \text{ km/h}$
   - Cible : $v_{B_{\text{proj}}} = \begin{pmatrix} 85 \\ 85 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 85 \text{ km/h}$
4. **Conclusion** : La police fonce vers le Nord à 100. La cible s'éloigne vers le Nord à 85 seulement (le reste de sa vitesse part vers l'Est).

$$v_{A_{\text{proj}}} > v_{B_{\text{proj}}}$$

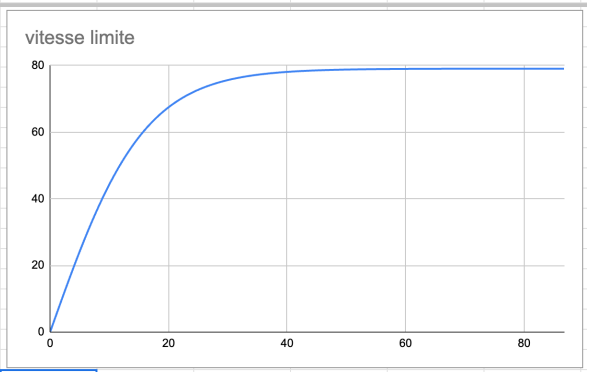
   Réponse : Oui, la police rattrape la cible (à une vitesse relative de 15 km/h). Mais attention, cela n'est vrai qu'au moment du scan. Comme la voiture de la cible se déplace vers l'est, le vecteur de visée change constamment.

Thème 2 : Forces, Gravité et Frottements

Solutions et Logique Code.

Solution 4 : Le Drag Race

	temps qui passe	force moteur	force frottement	masse	acceleration	vitesse	distance
2	0	5000	0	1000	5	0	0
3	0,1	5000	0,2	1000	4,9998	0,5	0,05
4	0,2	5000	0,7999680003	1000	4,999200032	0,99998	0,149998
5	0,3	5000	1,799760016	1000	4,99820024	1,499900003	0,2999880003
6	0,4	5000	3,19910415	1000	4,996800896	1,999720027	0,499960003
7	0,5	5000	4,997600755	1000	4,995002399	2,499400117	0,7499000147
8	0,6	5000	7,19472268	1000	4,992805277	2,998900357	1,04979005
9	0,7	5000	9,7898156	1000	4,990210184	3,498180884	1,399608139
10	0,8	5000	12,78209844	1000	4,987217902	3,997201903	1,799328329
11	0,9	5000	16,17066388	1000	4,983829336	4,495923693	2,248920698
12	1	5000	19,95447894	1000	4,980045521	4,994306627	2,748351361
13	1,1	5000	24,13238567	1000	4,975867614	5,492311179	3,297582479
14	1,2	5000	28,70310187	1000	4,971296898	5,98989794	3,896572273
15	1,3	5000	33,66522198	1000	4,966334778	6,48702763	4,545275036
16	1,4	5000	39,01721797	1000	4,960982782	6,983661108	5,243641147
17	1,5	5000	44,75744038	1000	4,95524256	7,479759386	5,991617085
18	1,6	5000	50,88411934	1000	4,949115881	7,975283642	6,78914545
19	1,7	5000	57,39536579	1000	4,942604634	8,47019523	7,636164973



Physique : La « Vitesse Terminale » est atteinte quand l'accélération est nulle, donc quand les forces s'annulent.

$$\sum F = 0 \text{ implique } F_{\text{moteur}} + F_{\text{air}} = 0$$

$$5000 - kv^2 = 0$$

$$kv^2 = 5000$$

Calcul :

$$v = \sqrt{\frac{5000}{0.8}} = \sqrt{6250} \approx 79.05 \text{ m/s}$$

Soit environ 284 km/h.

Solution 5 : Le Sniper

1. Décomposition de la vitesse initiale ($t = 0$)

Avec $v_0 = 50$ m/s et $\theta = 45°$:

$$v_{\{0x\}} = v_0 \cos(\theta) = 50 \times 0.707 \approx 35.35 \text{ m/s}$$

$$v_{\{0y\}} = v_0 \sin(\theta) = 50 \times 0.707 \approx 35.35 \text{ m/s}$$

2. Équations de position (au temps t)

On néglige les frottements. L'accélération ne joue que sur la hauteur (y) :

- Horizontal : $x(t) = v_{\{0x\}} \times t$
- Vertical : $y(t) = v_{\{0y\}} \times t - \frac{1}{2}gt^2$

3. Calcul du temps de vol ($t_{\{\text{vol}\}}$)

Le projectile touche le sol quand $y(t) = 0$ (pour $t > 0$) :

$$35.35t - 0.5 \times 9.81 \times t^2 = 0$$

$$t \times (35.35 - 4.905t) = 0$$

$$t_{\{\text{vol}\}} = \frac{35.35}{4.905} \approx 7.207 \text{ secondes}$$

4. Calcul de la portée (R)
C'est la distance parcourue horizontalement pendant le temps de vol :

$$R = x\left(t_{\{\text{vol}\}}\right) = 35.35 \text{ m/s} \times 7.207\text{s} \approx 254.8 \text{ mètres}$$

Note Code : Si votre simulation (Euler) donne une valeur éloignée (ex: 260m), c'est que votre pas de temps Δt est trop grand pour capter précisément l'instant $t_{\{\text{vol}\}}$.

Solution 6 : Le Cube (Logique Code)

Algorithme à implémenter dans `updatePhysics` :

```
// 1. Calcul des forces
const Px = mass * g * Math.sin(angle); // Force qui tire vers le bas
const Py = mass * g * Math.cos(angle); // Force qui plaque au sol

const frictionMax = mu * Py; // Résistance max du sol

// 2. Décision (Seuil)
if (Px > frictionMax) {
    // Glissement : La force nette est la différence
    const netForce = Px - (mu_cinetique * Py);
    acceleration = netForce / mass;
} else {
    // Statique : Pas de mouvement
    acceleration = 0;
    velocity = 0;
}
```

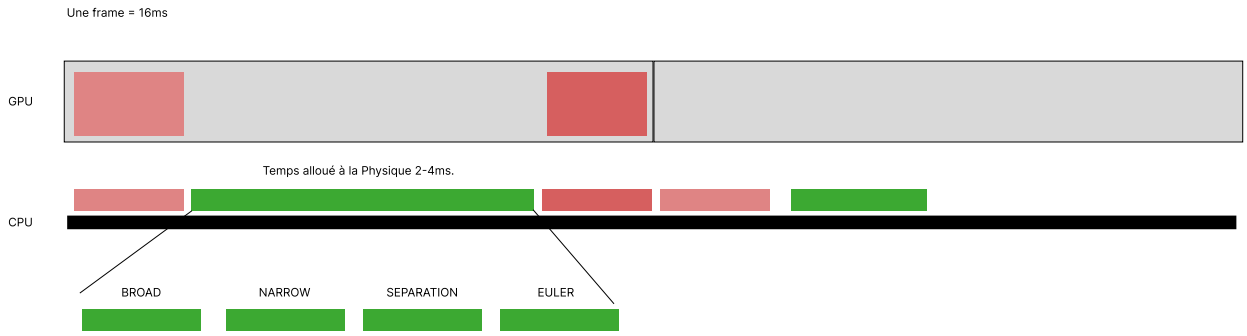
Session 6 : Broad & Narrow Phases

1. Le Contexte : Le Budget Temps (16ms)

Dans un jeu vidéo tournant à 60 FPS, vous avez 16.6 millisecondes pour **tout** calculer.

La physique n'est qu'une part du gâteau :
- Rendu (Graphics) : 8ms
- IA / Gameplay : 4ms
- Audio / Réseau : 2ms
- **Physique : 2ms**

Si votre détection de collision prend 10ms, le jeu saccade (« lag »). L'optimisation n'est pas un bonus, c'est une nécessité vitale.



2. Le Problème Mathématique : La Malédiction O(N²)

Pourquoi la détection « naïve » échoue-t-elle ? Imaginons une matrice d'interaction où chaque objet (Ligne) teste chaque objet (Colonne).

Pour $N = 5$ objets (A, B, C, D, E) :

	A	**B**	**C**	**D**	**E**
A	X	Test	Test	Test	Test
B	Skip	X	Test	Test	Test
C	Skip	Skip	X	Test	Test
D	Skip	Skip	Skip	X	Test
E	Skip	Skip	Skip	Skip	X

- La Diagonale (X) : Test inutile (A vs A).
- Le triangle inférieur (Skip) : Redondant (si A touche B, B touche A).

Le Coût Réel :

$$C \approx \frac{N(N-1)}{2} \approx O(N^2)$$

- 10 objets → 45 tests (Négligeable)
- 1 000 objets → 499 500 tests (Lent)
- 10 000 objets → 50 000 000 tests (Crash CPU)

Solution : Il faut réduire N avant de faire les tests coûteux. C'est le rôle de la Broad Phase.

3. La Broad Phase : Structures de Partitionnement

Le but est de répondre rapidement à la question : **« Qui sont mes voisins ? »**. On remplace les objets complexes par des AABB (Axis Aligned Bounding Box).

A. Grille Uniforme (Spatial Hashing)

- Concept : On découpe le monde en cases de taille fixe. On ne teste l'objet A que contre les objets dans la même case (ou les cases adjacentes).
- Avantage : Accès $O(1)$, très rapide, facile à coder.
- Inconvénient : Mauvais si le monde est infini ou si les objets ont des tailles très variées.

B. Arbres Spatiaux (Quadtree / Octree)

- Concept : Découpage récursif. Si une case contient trop d'objets, on la coupe en 4 (2D) ou 8 (3D).
- Avantage : S'adapte à la densité (beaucoup de cases là où il y a des objets, peu ailleurs).
- Inconvénient : Coûteux à mettre à jour si les objets bougent beaucoup.

C. Sweep and Prune (SAP)

- Concept : On projette les débuts/fins des AABB sur les axes X, Y, Z et on trie la liste. Si les intervalles sur X ne se chevauchent pas, inutile de tester Y et Z.
- Secret : Exploite la **Cohérence Temporelle** (les objets ne se téléportent pas, la liste reste presque triée d'une frame à l'autre).

4. La Narrow Phase : Les Méthodes de Résolution

On a filtré 99% des cas. Il reste les paires « proches ». Comment savoir si elles se touchent vraiment ? Deux écoles dominent :

Méthode 1 : SAT (Separating Axis Theorem)

- Philosophie : « L'ombre ». Si je peux trouver une lumière qui projette des ombres séparées des deux objets, alors ils ne se touchent pas.
- Fonctionnement : On projette les formes sur une série d'axes (Normales des faces, Produits vectoriels des arêtes). Si **une seule** projection montre un écart, il n'y a pas collision.
- Utilisation : Idéal pour Boîtes vs Boîtes ou Polyèdres simples.

Méthode 2 : GJK (Gilbert-Johnson-Keerthi)

- Philosophie : « La Différence ». On travaille dans l'espace de Minkowski ($A - B$).
- Fonctionnement : On cherche itérativement si l'Origine $(0, 0, 0)$ est contenue dans la forme $A - B$ en construisant un **Simplex** (Tétraèdre) à l'intérieur.
- Utilisation : Le standard pour les formes convexes quelconques (Capsules, Cones, Mesh convexes).

Focus : Comprendre GJK et le Support Mapping

GJK est un algorithme « aveugle ». Il ne connaît pas la géométrie, il pose juste une question à l'objet via une Fonction de Support :

« Quel est ton point le plus extrême dans la direction $\vec{d}$? »

$$S_{A-B}(\vec{d}) = S_{A(\vec{d})} - S_{B(-\vec{d})}$$

Cela permet de découpler l'algorithme de collision (GJK) de la définition géométrique des objets.

- Sphère : $\text{Centre} + \text{Rayon} * \vec{d}$
- Cube : Coin correspondant au signe de $\vec{d}$

Résumé du Pipeline

1. **Update Position :** $P = P + V * \text{dt}$
2. **Broad Phase :** Mise à jour de la grille/Octree -> Liste de paires candidates.
3. **Narrow Phase :**
   - Check AABB (rapide)
   - Check GJK ou SAT (précis) -> Collision OUI/NON
4. **Contact Generation (EPA/Clipping) :** Point de contact, Normale, Profondeur.
5. **Resolution :** Application des impulsions pour séparer les objets.

Session presencielle - Révisions et Quiz (20 points)

Session 8 : Physique de la Rotation

L'objectif du jour

Passer de la particule (un point) au corps rigide (une forme). Un corps rigide ne fait pas que se déplacer, il **pivote** autour de son centre de masse.

1. Le Dictionnaire de Traduction

Concept	Translation (Linéaire)	Rotation (Angulaire)
Position	$\vec{r}$ (m)	Angle θ (rad)
Vitesse	$\vec{v}$ (m/s)	Vitesse Angulaire ω (rad/s)
Résistance	Masse m (kg)	Moment d'Inertie I (kg·m²)
Cause	Force $\vec{F}$ (N)	Couple / Torque τ (N·m)
Loi de Newton	$\vec{F} = m\vec{a}$	$\tau = I\alpha$

2. Le Couple (Torque) : Faire tourner

Le couple τ dépend de où on pousse.

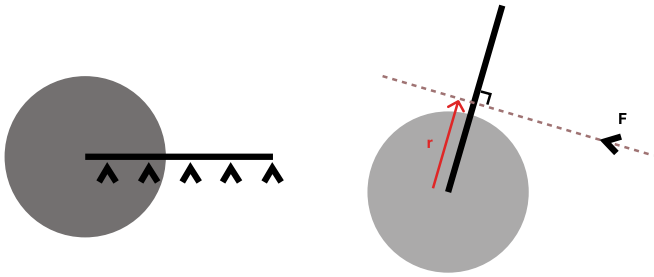


Fig. 12. – Plus on tire loin du centre, plus l'effet de rotation est fort.

En 3D (Vectoriel) :

La formule générale utlise un produit vectoriel.

$$\vec{\tau} = \vec{r} \times \vec{F}$$

τ est donc un vecteur perpendiculaire au plan de rotation.

En 2D (Scalaire) :

En 2D, la formule se simplifie.

$$\tau = r_x F_y - r_y F_x$$

(Où $\vec{r}$ est le vecteur allant du centre de masse au point d'impact)

3. Le Moment d'Inertie (I ou J)

Le moment d'inertie représente la résistance d'un objet à la rotation.

Il dépend de la répartition de la masse, et de la forme de l'objet.

Toujours par rapport a l'axe de rotation bien entendu.

La formule générale est :

$$\boldsymbol{I = \sum_i m_i.r_i^2}$$

ou r_i est la distance au centre de rotation de la masse m_i.

Pour un anneau mince de masse m et de rayon R, on a :

$$I = mR^2$$

, en effet, toute la masse de l'anneau est à la distance fixe R du centre de rotation.

Exercice A : Calcul du moment d'inertie d'un anneau

Retrouver $I = mR^2$ pour un anneau mince de masse m et de rayon R.

1. Paramètres :
- Densité linéaire : $\lambda = \frac{m}{L} = \frac{m}{2\pi R}$ (où L est la circonférence).
- Masse infinitésimale d'un petit segment : $dm = \lambda R d\theta$.
- Distance à l'axe pour tout point : $r = R$.

2. Calcul par intégration : Le moment d'inertie total est la somme (intégrale) des moments infinitésimaux $dI = r^2 dm$.

$$I = \int dI = \int_0^{2\pi} R^2 \cdot (\lambda R d\theta)$$

Sortons les constantes de l'intégrale :

$$I = \lambda R^3 \int_0^{2\pi} d\theta$$

Résolution de l'intégrale :

$$I = \lambda R^3 [\theta]_0^{2\pi}$$

$$I = \lambda R^3 (2\pi - 0) = 2\pi\lambda R^3$$

3. Substitution et simplification : Remplaçons λ par sa définition ($\frac{m}{2\pi R}$) :

$$I = \left(\frac{m}{\cancel{2\pi R}}\right) \cdot \cancel{2\pi} R^{\cancel{2}}$$

$$\boxed{\boldsymbol{I = mR^2}}$$

> **Conclusion :** Pour un anneau, toute la masse est à la distance maximale R. C'est l'objet qui offre la plus grande résistance à la rotation pour une masse donnée.

- Rectangle (Largeur w, Hauteur h) :

$$I = \frac{1}{12}m(w^2 + h^2)$$

Pourquoi $\frac{1}{12}$? C'est le résultat mathématique de l'intégration de la masse sur toute la surface du rectangle.

4. Vitesse d'un point sur le corps

C'est l'équation la plus importante pour la détection de collision future. Si un corps se déplace à $\vec{v}_G$ et tourne à ω, la vitesse d'un point P situé à la distance $\vec{r}$ du centre est :

$$\vec{v}_P = \vec{v}_G + (\omega \times \vec{r})$$

En 2D, cela devient :

$$v_{P.x} = v_{G.x} - \omega \cdot r_y$$

$$v_{P.y} = v_{G.y} + \omega \cdot r_x$$

> **Travaux Pratiques : Le 'Spinning Box'**

> **Objectif**
>
> Implémenter un rectangle qui tourne lorsqu'on clique dessus avec la souris (application d'une force à un point précis).

Étape 1 : Mise à jour de la classe `RigidBody`

Ajoutez les propriétés suivantes à vos objets :

- `angle` (float)
- `angularVelocity` (float)
- `torque` (float)
- `Inertia` (float) : Calculez-le une seule fois au début via la formule du rectangle.

Étape 2 : La méthode `ApplyForce(force, worldPoint)`

Cette méthode remplace l'ancien ajout direct à l'accélération :

1. Calculer le bras de levier : $\vec{r} = \text{worldPoint} - \text{positionCentre}$.
2. Ajouter à la force linéaire : $\vec{F}_{\text{total}} += \overrightarrow{\text{force}}$.
3. Ajouter au torque : $\tau += \left(r_x F_y - r_y F_x\right)$.

Étape 3 : L'intégrateur (Euler Semi-Implicite)

Dans votre boucle `update(dt)` :

1. $a = \frac{F}{m}$
2. $\alpha = \frac{\tau}{I}$
3. $v += a \cdot dt$
4. $\omega += \alpha \cdot dt$
5. $p += v \cdot dt$
6. $\theta += \omega \cdot dt$
7. Important : Remettre `F` et `tau` à zéro pour la frame suivante.

> **Test de compréhension**
>
> « A) L'objet tourne plus vite si je clique au centre. », « B) L'objet tourne plus vite si je clique sur un coin. », « C) L'endroit du clic n'a pas d'importance. »

« Pensez au bras de levier r. »

Session de soutien pour le TP1

Session 10 : Encore plus de forces

Introduction

Jusqu'à présent, nos particules étaient soit libres, soit en collision. Aujourd'hui, nous allons voir d'autres types de forces. Dans un premier temps, nous allons voir comment **lier** des particules ensemble. Cela va nous permettre de créer des objets plus complexes, comme des chaînes, des tissus, ou des structures, et d'avoir des comportements plus intéressants. Dans un deuxième temps, nous allons mettre en pratique les forces gravitationnelles pour un TP épique.

Il existe deux manières de voir un lien :
1. Comme une **Force** qui tire (Loi de Hooke).
2. Comme une **Règle** géométrique à respecter (Contrainte de Verlet).

1. La Loi de Hooke (L'approche par Force)

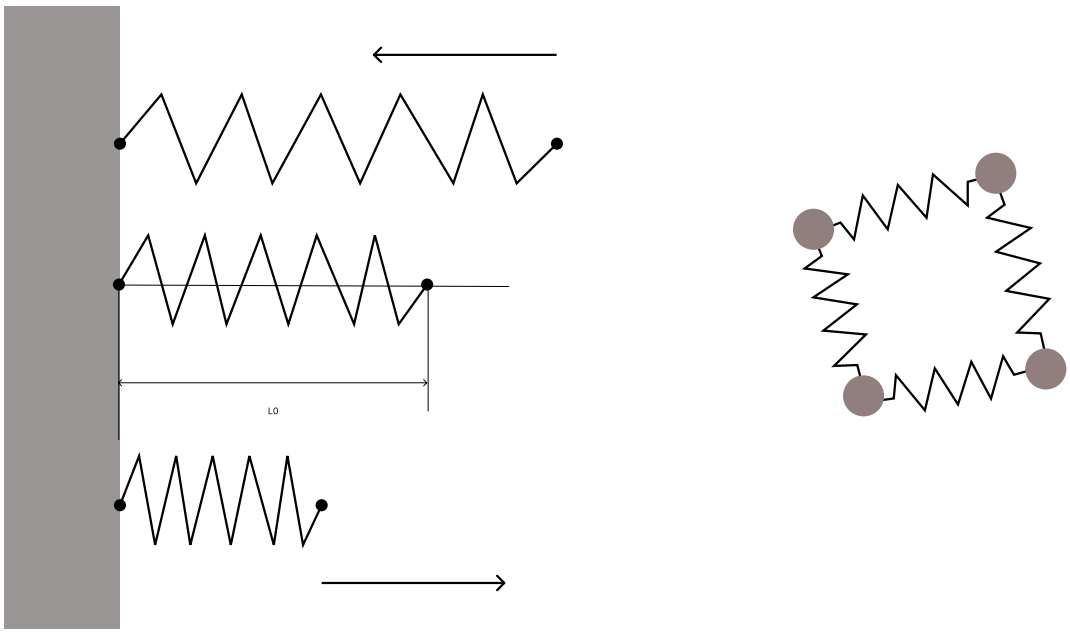


Fig. 13. – loi de Hook

Définition

La loi de Hooke modélise la **force exercée par un ressort**. La force est proportionnelle à l'étirement (ou la compression) du ressort.

$$\vec{F}_s = -k \cdot (|L - L_0|) \cdot \hat{u}$$

- k : Constante de raideur (stiffness). Plus k est grand, plus le ressort est dur.
- L : Distance actuelle entre les deux points.
- L_0 : Longueur au repos (rest length).
- $\hat{u}$: Vecteur unitaire de la direction du ressort.

C'est une loi linéaire (qui est proportionnelle à la distance) qui décrit le comportement d'un ressort idéal.

On peut noter que:
- lorsque $L = L_0$, la force est nulle.
- lorsque $L > L_0$, la force est négative (le ressort est étiré).
- lorsque $L < L_0$, la force est positive (le ressort est comprimé).

Cela signifie que la force est toujours dirigée vers la position d'équilibre.

Le problème de la stabilité

En utilisant Euler avec la loi de Hooke, si k est trop élevé ou si le dt est trop grand, le système accumule de l'énergie et explose. Pour stabiliser un ressort, il faut ajouter une Force d'Amortissement (Damping) pour dissiper l'énergie parasite :

$$\vec{F}_d = -c \cdot (\vec{v}_\text{A} - \vec{v}_\text{B})$$

2. Contraintes de Distance (L'approche Verlet)

La méthode géométrique

Avec l'intégration de Verlet, au lieu de calculer une force, on déplace directement les points pour satisfaire la longueur souhaitée. C'est un système de relaxation.

Algorithme pour une contrainte entre A et B :
1. Calculer la distance actuelle $d = \|P_\text{A} - P_\text{B}\|$.
2. Calculer l'erreur relative : $e = \frac{d - L_0}{d}$.
3. Déplacer A et B de la moitié de l'erreur :

$$P_\text{A} = P_\text{A} - 0.5 \cdot e \cdot (P_\text{A} - P_\text{B})$$

$$P_\text{B} = P_\text{B} + 0.5 \cdot e \cdot (P_\text{A} - P_\text{B})$$

La Rigidité par l'itération

Si vous n'appliquez cette correction qu'une seule fois par frame, la corde semblera élastique. Pour obtenir un câble rigide, il faut répéter cette opération plusieurs fois par frame (5 à 15 itérations). C'est ce qu'on appelle la Relaxation de Gauss-Seidel.

3. Travaux Pratiques : Le système solaire

Objectif

Créer une simulation du système solaire.

La force gravitationnelle

La force gravitationnelle est une force d'attraction universelle qui s'exerce entre tous les corps possédant une masse, s'attirant mutuellement sans se toucher. Son intensité dépend directement des masses des objets et diminue avec le carré de la distance qui les sépare (selon la célèbre loi de la gravitation universelle de Newton).

C'est cette force qui maintient la Lune en orbite autour de la Terre et qui donne leur poids aux objets sur Terre

Elle est définie par la loi de la gravitation **universelle** de Newton :

$$F = G \cdot \frac{m_1 m_2}{r^2}$$

où F représente la force d'attraction entre les corps (Newton)

G représente la constante de la gravitation universelle ($6.67 \times 10^{-11} N \cdot \frac{m^2}{kg^2}$)

m_1 représente la masse du premier objet (kg)

m_2 représente la masse du deuxième objet (kg)

r représente la distance séparant les deux objets (m)

Le défi du TP : L'orbite stable

Pour qu'une planète orbite autour d'une étoile sans s'écraser ni s'enfuir, elle doit posséder une **vitesse tangentielle** initiale précise.

$$v_{\text{orbitale}} = \sqrt{\frac{G \cdot M_{\text{étoile}}}{r}}$$

Données astronomiques

Nous utiliserons des valeurs relatives.

Le Soleil sera placé au centre avec une masse de 333 000 (unités terrestres).

Astre	**Masse relative** (Terre = 1)	**Distance** (UA)	**Vitesse Orbitale** (Relative)
Mercure	0.055	0.39	1.60
Vénus	0.815	0.72	1.17
Terre	1.000	1.00	1.00

Astre	**Masse relative** (Terre = 1)	**Distance** (UA)	**Vitesse Orbitale** (Relative)
Mars	0.107	1.52	0.80
Jupiter	317.8	5.20	0.43
Saturne	95.2	9.54	0.32
Uranus	14.5	19.20	0.22
Neptune	17.1	30.10	0.18

Conseils pour l'échelle du TP

- **Unités :** Si vous prenez 1 UA = 100 pixels, Neptune sera trop loin (3000 px). Essayez plutôt 1 UA = 20 pixels ou utilisez un zoom caméra.
- **Vitesse initiale :** Pour que la Terre reste en orbite circulaire, sa vitesse de départ doit être perpendiculaire au rayon Soleil-Terre.
- **G :** Réglez votre constante G de sorte que la Terre mette environ 10 secondes à faire un tour complet.

Vitesse d'orbite

Pour qu'une planète reste en orbite circulaire, sa vitesse initiale doit être perpendiculaire au rayon Soleil-Terre.

Sa vitesse doit être :

$$v = \sqrt{\frac{G \cdot M_{\text{soleil}}}{r}}$$

Démonstration

Dans la session 2 sur la cinématique, nous avons vu que la magnitude de l'accélération centripète est :

$$a = \|\vec{a}(t)\| = \omega^2 R$$

Comme la vitesse est $v = \omega R$, on peut exprimer l'accélération en fonction de v :

$$\omega = \frac{v}{R} \longrightarrow a = \left(\frac{v}{R}\right)^2 \cdot R = \frac{v^2}{R}$$

1. La force requise (Newton) : Pour maintenir ce mouvement circulaire, la nature doit fournir une force :

$$F_c = m \cdot a = \frac{mv^2}{R}$$

2. La force fournie (Gravitation) : Dans l'espace, c'est la gravité qui joue ce rôle :

$$F_g = \frac{GMm}{R^2}$$

3. L'équilibre orbital : En égalisant la force requise et la force fournie ($F_c = F_g$) :

$$\frac{mv^2}{R} = \frac{GMm}{R^2}$$

En simplifiant par la masse de la planète (m) et par un rayon (R), on obtient :

$$v^2 = \frac{GM}{R}$$

D'où la vitesse orbitale circulaire utilisée dans notre code :

$$\boldsymbol{v = \sqrt{\frac{GM}{R}}}$$

Session 11 : Moteurs Physiques & Middleware

Introduction

Félicitations ! Vous avez écrit votre propre moteur physique basé sur Euleur puis Verlet. Vous comprenez maintenant la complexité cachée derrière de simples balles qui rebondissent : intégration, détection (Broad/Narrow phase) et résolution.

Dans l'industrie (Jeux Vidéo, Cinéma, Simulation), nous utilisons rarement des moteurs faits maison (« Homebrew ») pour la physique complexe. Nous utilisons des **Middlewares**.

Cette session explore les géants du marché, leurs spécificités, et comment choisir le bon outil pour Raylib ou Three.js.

1. Concepts avancés des moteurs commerciaux

Avant de comparer les moteurs, il faut comprendre les problèmes qu'ils résolvent et que notre petit moteur Verlet ne gère pas encore.

Rigid Body vs Soft Body

- **Rigid Body (Corps Rigide) :** L'objet ne se déforme jamais. La distance entre deux atomes de l'objet reste constante. C'est 95% de la physique dans les jeux (caisses, personnages, voitures).
- **Soft Body (Corps Mou) :** L'objet se déforme (tissus, gelée, pneus). C'est ce que nous avons simulé avec les ressorts et Verlet ! C'est beaucoup plus coûteux en calcul (CPU).

Le problème du Tunneling (CCD)

Dans notre moteur, si une balle va très vite, elle peut traverser un mur entre deux frames car P_{old} est d'un côté et P_{new} de l'autre.

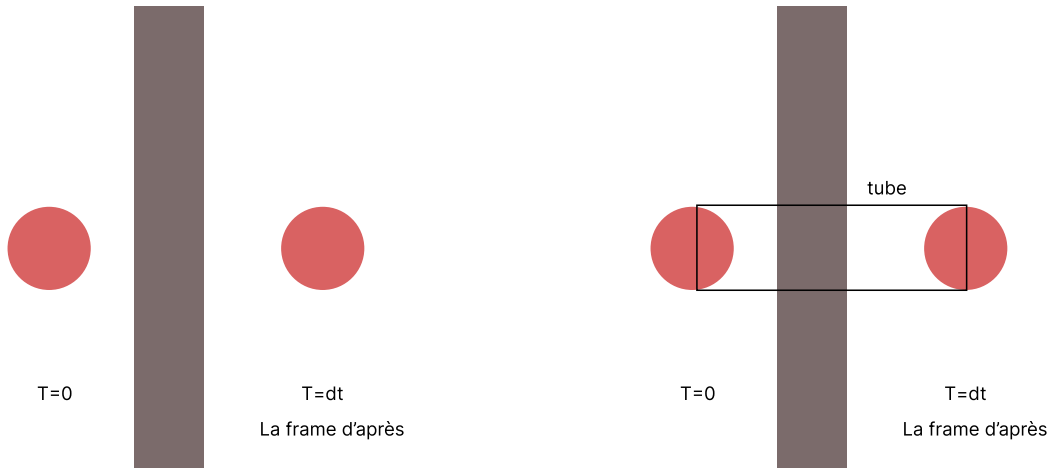


Fig. 14. – CCD (Continuous Collision Detection) pour éviter le tunneling

Les moteurs commerciaux utilisent le **CCD (Continuous Collision Detection)**. Au lieu de regarder des positions discrètes, ils extrudent le volume de l'objet le long de son vecteur vitesse pour vérifier s'il a « balayé » un obstacle.

Sleeping (L'endormissement)

Pour optimiser, un moteur physique détecte quand un objet arrête de bouger (vitesse < seuil pendant x frames). Il le met en état de **Sleep**. L'objet est retiré des calculs d'intégration jusqu'à ce qu'une force extérieure le réveille.

Sans cela, un empilement de 1000 caisses consommerait 100% du CPU même immobile.

2. Les Géants du C++ (Industrie AAA)

Ces moteurs sont les standards de l'industrie pour les jeux PC et Consoles.

Moteur	Spécificités
PhysX (NVIDIA)	• Le standard actuel (utilisé par Unity et Unreal Engine jusqu'à récemment). • Très rapide, très stable. • Spécificité : Peut utiliser la carte graphique (GPU) pour les particules et fluides complexes.
Havok	• Le vétéran (Half-Life 2, Zelda Breath of the Wild). • Incroyablement robuste pour les **Character Controllers** (mouvements de personnages). • Très cher (licence propriétaire historique), mais performant.
Jolt Physics	• La nouvelle star (Open Source). • Utilisé dans **Horizon Forbidden West**. • **Spécificité :** Conçu pour le **Multithreading** massif. Il écrase les autres sur les processeurs modernes à plusieurs cœurs.

3. Le Roi de la 2D : Box2D

Box2D

Si vous avez joué à **Angry Birds**, **Limbo** ou **Shovel Knight**, vous avez utilisé **Box2D**

Créé par Erin Catto (Blizzard), c'est un moteur « Impulse Based » (comme notre résolution de collision, mais plus mathématique avec des matrices de masse).

C'est la référence absolue pour la 2D. Il est déterministe, stable et gère parfaitement les « stacks » (piles d'objets).

4. Physique pour le Web (Three.js / Javascript)

Puisque nous travaillons parfois avec le Web, le choix du moteur est critique car le JavaScript est moins performant que le C++.

Moteur	Techno	Verdict
Cannon.js	Pur JS	• Facile à utiliser avec Three.js. • Lent si > 100 objets. • Abandonné, mais fork « Cannon-ES » actif.
Ammo.js	WASM (C++)	• C'est le moteur « Bullet » compilé en WebAssembly. • Très complet mais lourd à charger. • API complexe et documentation difficile.
Rapier	Rust -> WASM	• Le choix moderne. • Écrit en Rust, compilé pour le web. • Extrêmement rapide et stable. • API moderne et bien documentée.

Pourquoi Rapier ?

Si vous devez choisir un moteur aujourd'hui pour un projet Web/Three.js, choisissez Rapier. Sa structure de données (SIMD) permet de gérer des milliers d'objets sans ralentir le navigateur, là où Cannon.js s'effondrerait.

5. Déterminisme vs « Gameplay »

Le Déterminisme

Une simulation est dite déterministe si : **« Pour un état initial donné et un delta time fixe, le résultat est strictement identique bit-à-bit sur n'importe quel ordinateur. »**

C'est vital pour :

1. Le **Multijoueur** (Pour que tous les joueurs voient la même caisse tomber au même endroit).
2. Les **Replays** (Pour rejouer une partie enregistrée).

Les flottants (float) ne sont pas gérés pareil par tous les processeurs (Intel vs AMD vs ARM). Les moteurs comme **Box2D** ou des versions « Fixed Point » s'efforcent d'être déterministes.

Conclusion : Faut-il faire son propre moteur ?

- **OUI** pour apprendre (ce cours).
- **OUI** pour des cas très simples (Pong, particules visuelles sans gameplay).
- **OUI** pour des cas très spécifiques (Simulation de cordes pure, simulation de sable).
- **NON** pour un jeu commercial complexe. La gestion des collisions 3D (Mesh vs Mesh), l'optimisation spatiale (Octrees/BVH) et la stabilité numérique sont des années de travail que PhysX ou Rapier vous offrent gratuitement.

6. Anatomie d'un Moteur : Architecture ECS

Pourquoi cette complexité ?

Dans notre moteur Verlet, nous avions un `vector<Ball>`. C'est simple, mais ça passe mal à l'échelle (cache CPU, mémoire).

Les moteurs modernes (Rapier, Jolt, Unity ECS) séparent strictement les **Données** de la **Logique**.

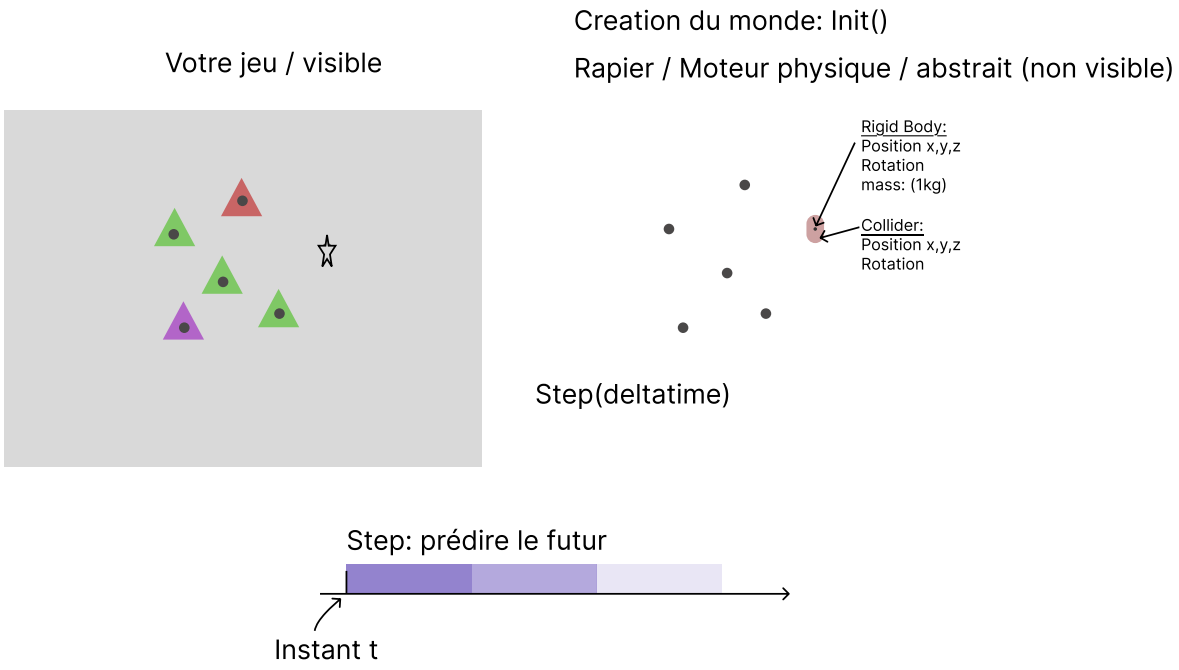


Fig. 15. – Séparation des données et de la logique, un seul objectif: prédire le futur

Pour être efficace, ils utilisent une architecture basée sur les **Entités, Composants et Systèmes** (ECS). En effet, ces architectures **permettent d'optimiser et paralléliser** les taches répétitives simples sur de nombreux objets (comme le calcul de la physique).

A. Les Structures de Données (The Sets)

Au lieu d'avoir un objet `Ball` qui contient tout (Position, Forme, Masse), Rapier découpe les données dans des conteneurs spécialisés appelés Sets. On n'accède pas aux objets par pointeur, mais par Handle (un ID unique).

- RigidBody Set : Stocke la position (x, y, z), la vitesse et les forces. C'est « l'âme » physique de l'objet.
- Collider Set : Stocke la forme géométrique (Cube, Sphère) et les matériaux (Friction, Restitution). C'est le « corps » tangible.
- Joint Set : Stocke les contraintes (Ressorts, Pivots) qui relient deux RigidBodies.

> **Generational Arena**
>
> Ces « Sets » ne sont pas de simples tableaux. Ce sont des **Generational Arenas**. Si vous supprimez l'objet ID `42` et en créez un nouveau qui prend la place `42`, le moteur ajoute une « génération » (ex: `42_v2`). Cela empêche les bugs où une balle continuerait de suivre un objet détruit.

B. Le Pipeline Physique

La fonction `step()` n'est pas magique. Elle orchestre plusieurs étapes critiques séquentiellement :

1. **Gravity & Forces :** Application de la gravité globale et des forces utilisateur.
2. **Broad Phase :** Trouve grossièrement qui **pourrait** toucher qui (AABB).
3. **Narrow Phase :** Calcule précisément les points de contact.
4. **Island Manager (Optimization) :** Le moteur regroupe les objets qui se touchent en « Îles ».
   - Si tous les objets d'une île ont une vitesse quasi-nulle, l'île entière est mise en Sleep (Sommeil).
   - Le moteur cesse de les calculer jusqu'à ce qu'une force extérieure les réveille.
   - **C'est le secret pour afficher 10 000 caisses à 60 FPS.**
5. **Solver :** Résout les contraintes (empêcher la pénétration, gérer les joints).
6. **Integration :** Met à jour les positions finales (Euler: $P = P + V \cdot dt$).

C. Les Pipelines Spécialisés

Tous les calculs ne servent pas à faire bouger des objets. Parfois, on veut juste poser une question au monde.

> **Query Pipeline (Interrogation)**
>
> C'est le module utilisé pour le Gameplay (comme notre tir de catapulte). Il utilise les structures accélérées (BVH - Bounding Volume Hierarchy) pour répondre instantanément à :
> - Raycasting : « Qu'est-ce qui est sous ma souris ? »
> - Shapecasting : « Si j'avance ce personnage ici, va-t-il cogner un mur ? »
> - Point Projection : « Quel est l'objet le plus proche de cette bombe ? »

D. Événements et Hooks

Parfois, la physique ne suffit pas, il faut du code de jeu.

- Physics Hooks : Permet de filtrer les collisions **avant** qu'elles ne soient résolues (ex: « Les gentils traversent les gentils, mais cognent les méchants »).

- Event Handler : Notifie le jeu **après** une collision (ex: « Si la balle touche le mur à haute vitesse -> Jouer un son “Boom” »).

Session 12 : RigidBodies & Colliders

> **Introduction**
>
> Nous passons maintenant de la théorie (écrire notre propre moteur) à la pratique professionnelle (utiliser un middleware).
>
> Pour les projets Web/Three.js, nous utiliserons **Rapier** (écrit en Rust, compilé en WASM). Pour les projets C++/Raylib, nous utiliserons **Box2D** ou **PhysX**.
>
> Les concepts sont identiques à 99% entre tous ces moteurs.

1. Le Monde (Physics World)

Tout commence par la création d'un monde. C'est le conteneur qui gère la gravité et la liste des corps.

Initialisation (Rapier).

```
import RAPIER from '@dimforge/rapier3d-compat';

await RAPIER.init(); // Charger le WASM
let gravity = { x: 0.0, y: -9.81, z: 0.0 };
let world = new RAPIER.World(gravity);
```

2. RigidBody : L'Existence Physique

Un objet physique est séparé en deux concepts : le **Corps** (sa physique) et la **Forme** (sa collision).

> **Les 3 Types de Corps**
>
> 1. **Dynamic :** Soumis aux forces et aux collisions (Caisse, Balle, Joueur). C'est ce que nous avons fait jusqu'ici:
>
> $$F = m.a$$
>
> .
> 2. **Static (ou Fixed) :** Masse infinie, vitesse nulle. Ne bouge JAMAIS (Sol, Mur, Maison).
> 3. **Kinematic :** « Téléguidé ». Il a une aussi une masse infinie (il écrase tout) mais n'est pas affecté par les forces. On contrôle sa position manuellement soit par une animation, mais pas par la physique(Plateforme mobile, Ascenseur).

Création d'un corps.

```
// 1. Définir le corps
let rigidBodyDesc = RAPIER.RigidBodyDesc.dynamic()
    .setTranslation(0.0, 5.0, 0.0);

// 2. Créer le corps dans le monde
let rigidBody = world.createRigidBody(rigidBodyDesc);
```

3. Collider : La Forme de Collision

Le RigidBody n'a pas de forme. C'est juste un point avec une masse. Pour qu'il rebondisse, il faut lui attacher un **Collider**.

> **Collider vs Mesh**
>
> - **Mesh (Three.js) :** Ce qu'on **voit** (milliers de triangles, textures).
> - **Collider (Rapier) :** Ce qui **touche** (formes primitives simples).
>
> **Règle d'or :** Toujours utiliser la forme la plus simple possible pour le collider (Sphère, Cube, Capsule) pour les performances.

Attacher une forme.

```
// Une boite de 1x1x1 (demi-extents = 0.5)
let colliderDesc = RAPIER.ColliderDesc.cuboid(0.5, 0.5, 0.5)
    .setRestitution(0.7)  // Rebond
    .setFriction(0.5);    // Frottement

world.createCollider(colliderDesc, rigidBody);
```

4. La Boucle de Simulation

Comme pour notre moteur, il faut avancer le temps à chaque frame.

> **Synchronisation Visuelle**
>
> Rapier calcule la physique, mais ne dessine rien. C'est à vous de copier la position du corps physique vers le mesh visuel à chaque frame.

```
function animate() {
    requestAnimationFrame(animate);

    // 1. Avancer la physique
    world.step();

    // 2. Copier les positions (Physique -> Graphique)
    mesh.position.copy(rigidBody.translation());
    mesh.quaternion.copy(rigidBody.rotation());

    renderer.render(scene, camera);
}
```

5. Travaux Pratiques

Voir le sujet « Siege Engine - Partie A ». Vous devez construire un mur stable qui s'effondre de manière réaliste.

Session 13 : Joints, Moteurs & Raycasts

> **Introduction**
>
> Pour le moment, nous avons des briques qui tombent. C'est amusant, mais pour faire un vrai jeu (ou une machine complexe), nous devons relier les objets entre eux.
>
> C'est le rôle des **Joints** (Articulations) et des **Constraints**.

1. Les Joints (Articulations)

Un Joint relie deux RigidBodies (A et B) et limite leurs mouvements relatifs.

> **Les Types Principaux**
>
> - **Fixed Joint :** Colle deux objets ensemble (ex: souder deux pièces).
> - **Revolute Joint (Hinge) :** Une charnière. Rotation libre autour d'un axe (ex: Roue de voiture, Porte, Genou).
> - **Prismatic Joint (Slider) :** Glissière. Translation libre sur un axe (ex: Ascenseur, Piston).
> - **Spherical Joint (Ball & Socket) :** Rotation libre sur tous les axes (ex: Épaule, Attelage de remorque).

> **Ancres (Anchors)**
>
> Pour définir un joint, il faut préciser **où** il s'accroche sur chaque objet.
> - Anchor_A : Point d'attache dans le repère local de A.
> - Anchor_B : Point d'attache dans le repère local de B.

Créer une Charnière (Rapier).

```
// On veut attacher une roue (bodyB) à une voiture (bodyA)
// L'axe de rotation est l'axe X (1, 0, 0)

let jointDesc = RAPIER.JointData.revolute(
    { x: 0.0, y: -0.5, z: 0.0 }, // Anchor A (sous la voiture)
    { x: 0.0, y: 0.0, z: 0.0 },  // Anchor B (centre de la roue)
    { x: 1.0, y: 0.0, z: 0.0 }   // Axe de rotation
);

world.createImpulseJoint(jointDesc, bodyA, bodyB);
```

2. Moteurs & Limites

Les joints ne sont pas forcément passifs. On peut les motoriser ou restreindre leur mouvement.

Contrôle actif

- **Limits :** Empêcher une porte de s'ouvrir à plus de 90 degrés. `joint.setLimits(-PI/2, PI/2)`
- **Motor (Velocity) :** Faire tourner une roue à vitesse constante.
  `joint.configureMotorVelocity(10.0, 50.0)` (Vitesse cible, Force max)
- **Motor (Position) :** Servo-moteur qui cherche à atteindre un angle précis.
  `joint.configureMotorPosition(targetAngle, stiffness, damping)`

3. Raycasting (Interaction)

Comment cliquer sur un objet 3D ? Comment savoir si mon personnage voit l'ennemi ? On lance un rayon invisible (**Ray**) et on demande au moteur physique ce qu'il touche.

Fig. 16. – Le Raycast part de la caméra et traverse le monde

Tirer un rayon.

```
let ray = new RAPIER.Ray(origin, direction);
let maxToi = 100.0; // Distance max
let solid = true;   // Considérer les objets pleins comme obstacles

let hit = world.castRay(ray, maxToi, solid);

if (hit != null) {
    // On a touché quelque chose !
    let collider = hit.collider;
    let point = ray.pointAt(hit.timeOfImpact); // Position précise

    console.log("Touché : ", collider.handle);
}
```

4. Travaux Pratiques

Voir le sujet « Siege Engine - Partie B ». Vous allez construire une catapulte fonctionnelle avec des joints et interagir avec la souris.

Session 14 : Les Solveurs de Contraintes

> **Contextualisation**
>
> Dans notre moteur Verlet, nous déplacions directement les positions pour satisfaire les contraintes ($P_{\text{new}} = P_{\text{old}} + \ldots$). C'est ce qu'on appelle du **Position Based Dynamics (PBD)**.
>
> Les moteurs comme Rapier ou Box2D utilisent une approche différente, plus rigide, basée sur la vitesse : les **Impulsions Séquentielles**.

1. Qu'est-ce qu'une Contrainte ?

Pour un moteur physique, tout est une contrainte. C'est une règle mathématique $C(x)$ que le système doit respecter.

- **Contact (Non-pénétration) :** La distance entre deux objets doit être positive ($C \geq 0$).
- **Joint (Charnière) :** La distance entre deux points d'ancrage doit être nulle ($C = 0$).
- **Joint (Moteur) :** La vitesse angulaire doit être constante ($\delta\omega = k$).

Le rôle du solveur est de trouver les forces (ou impulsions) minimales à appliquer pour que $C(x)$ soit satisfait à la fin de la frame.

2. Rappel de « L'Impulsion » (La méthode Rapier)

Au lieu de calculer des forces (qui sont instables sur un pas de temps discret), les moteurs modernes calculent des **Impulsions**. Comme nous l'avons fait avec notre moteur Euler. Une impulsion J est un changement immédiat de vitesse.

$$v_{\text{finale}} = v_{\text{initiale}} + \frac{J}{m}$$

L'objectif du solveur est de trouver le J exact pour que la vitesse relative au point de contact devienne nulle (ou rebondisse).

3. L'Algorithme : Projected Gauss-Seidel (PGS)

Dans votre TP1, mis en place un solveur de position. On voulait séparer les billes, mais quand elles sont toutes collées ensemble, il fallait itérer plusieurs fois pour les séparer complètement.

Résoudre toutes les contraintes d'un coup (Global Solver) demande d'inverser des matrices géantes. C'est trop lent pour le temps réel. L'industrie utilise une méthode **Itérative** extrêmement efficace: le **Projected Gauss-Seidel**.

Le concept des Impulsions Séquentielles. *Imaginez une table à 4 pieds sur un sol inégal.*

1. *Vous calez le pied A. Le pied B se lève.*
2. *Vous calez le pied B. Le pied C se lève.*
3. *Vous calez le pied C...*

*Si vous faites le tour de la table une seule fois, elle est bancale. Si vous faites le tour 10 fois (**Itérations**), elle finit par être stable partout.*

L'algorithme simplifié :

1. Pour chaque contact et chaque joint...
2. Calculer l'erreur de vitesse.
3. Calculer l'impulsion correctrice J.
4. Appliquer J immédiatement aux deux corps.
5. Répéter l'opération N fois (généralement 10 à 20 fois).

4. Les Secrets de la Stabilité

Pourquoi Rapier est-il plus stable que notre moteur Verlet ? Grâce à deux concepts avancés.

A. Warm Starting (Démarrage à chaud)

Le solveur est itératif. Il part d'une solution devinée pour s'approcher de la solution réelle.

- **Cold Start :** On suppose que l'impulsion nécessaire est 0. Il faut beaucoup d'itérations pour trouver la bonne force de support d'une pile.
- **Warm Start :** On utilise **l'impulsion de la frame précédente** comme point de départ.

Résultat : Une pile de caisses est stable immédiatement car le moteur « se souvient » de la force nécessaire pour la porter.

B. Sleeping (Endormissement)

Même avec le Warm Starting, des micro-erreurs de calcul (flottants) créent du « Jittering » (tremblement). L'Island Manager surveille l'énergie cinétique globale d'un groupe d'objets. Si elle passe sous un seuil ($E < \varepsilon$) pendant quelques frames :

- On force les vitesses à $(0, 0, 0)$.
- On désactive le solveur pour ces objets.
- L'objet passe en mode **Sleep**.

> **Comparatif des Solveurs**

Type	Avantages	Inconvénients
PBD (Notre Verlet)	Stable, impossible à « casser », facile à coder.	Aspect « mou » ou caoutchouteux. Gestion de la friction difficile.
Impulse Based (Box2D, Rapier)	Très rigide (Hard constraints), empilement stable, friction précise.	Peut exploser si dt varie ou si contraintes conflictuelles.
Penalty Based (Ressorts)	Gère bien les corps mous et tissus.	Instable pour les corps rigides (vibrations).

5. Paramétrer le Solveur (API)

Dans `Rapier` ou `Cannon.js`, vous avez souvent accès à ces paramètres dans `IntegrationParameters` :

- **dt (TimeStep) :** Doit être fixe (ex: $\frac{1}{60}$).
- **Velocity Iterations :** Nombre de passes pour résoudre les vitesses (collisions). Standard : 4 à 10.

- **Position Iterations :** Nombre de passes pour corriger le chevauchement (Pénétration). Standard : 2 à 5.
- **ERP (Error Reduction Parameter) :** À quelle vitesse corrige-t-on une erreur (ex: joint qui s'étire). Trop haut = ressort violent. Trop bas = joint mou.

Session 15 : Cinématique Inverse (IK)

> **FK vs IK**
>
> Jusqu'à présent, nous avons pensé en **FK (Forward Kinematics)** : « Je tourne l'épaule de 30°, donc le coude bouge, donc la main avance. » C'est facile à calculer ($P_{\text{enfant}} = M_{\text{parent}} \times P_{\text{local}}$) mais difficile à contrôler.
>
> L'**IK (Inverse Kinematics)** pose le problème inverse : « Je veux que la main soit à la position (x, y, z). Comment dois-je tourner l'épaule et le coude ? »

1. Le Problème mathématique

Une chaîne cinématique est composée d'os (segments rigides) et d'articulations (joints). Pour un bras humain, on veut atteindre une cible (Target) avec l'effecteur final (End Effector).

- **Cas simple :** Un bras à 2 segments en 2D. On peut le résoudre par trigonométrie (Théorème d'Al-Kashi / Loi des cosinus).
- **Cas complexe :** Un tentacule à 10 segments en 3D. La trigonométrie devient infernale. Il existe une infinité de solutions possibles.

Comme pour le solveur physique de la Session 14, nous allons utiliser une **approche itérative**.

2. Algorithme 1 : CCD (Cyclic Coordinate Descent)

C'est l'algorithme « robotique ». Il est très simple à comprendre et à coder.

Principe : On part du bout de la chaîne (le poignet) et on remonte vers la racine (l'épaule). Pour chaque articulation, on la fait tourner pour aligner l'effecteur final vers la cible.

1. On regarde le dernier os. On le tourne pour qu'il pointe vers la cible.
2. On regarde l'avant-dernier os. On le tourne pour que le bout de la chaîne pointe vers la cible.
3. On continue jusqu'à la racine.
4. On répète la boucle jusqu'à ce que l'erreur soit faible.

Avantage : Très stable, permet de mettre des limites d'angles (constraints) facilement. **Défaut :** Donne un mouvement un peu raide, robotique.

3. Algorithme 2 : FABRIK

Forward And Backward Reaching Inverse Kinematics. C'est l'algorithme utilisé dans la plupart des jeux modernes (Unity, Unreal) car il est rapide et visuellement naturel. Il ne travaille pas avec des angles, mais avec des **positions**, exactement comme notre moteur Verlet (Session 10) !

L'algorithme en 2 passes :

1. **Backward (De la cible vers la base) :**
   - On place l'effecteur final **sur** la cible.
   - On tire le point précédent pour qu'il soit à la bonne distance (longueur de l'os).
   - On remonte jusqu'à la racine.
   - (Problème : la racine s'est détachée du corps !)

2. **Forward (De la base vers la cible) :**
   - On remet la racine à sa position d'origine (sur l'épaule).
   - On tire le point suivant pour qu'il soit à la bonne distance.
   - On redescend jusqu'à l'effecteur.

En répétant ces deux passes, la chaîne se tend naturellement vers la cible.

4. IK et Physique : L'Animation Procédurale

C'est ici que la magie opère. Comment mélanger Rapier (Physique) et IK (Animation) ?

Le placement des pieds (Foot IK). *Dans un jeu, un personnage court sur un terrain accidenté.*

1. ***Raycast :*** *On lance un rayon physique depuis le genou vers le bas pour trouver le point d'impact exact du sol (via le `QueryPipeline` de la session 11).*
2. ***Cible IK :*** *On définit ce point d'impact (+ décalage semelle) comme la cible IK du pied.*
3. ***Résolution :*** *On utilise FABRIK pour plier le genou et la cheville afin que le pied touche ce point.*

> **Active Ragdolls**
>
> Pour aller plus loin (ex: **Gang Beasts**, **Human Fall Flat**), on n'utilise pas l'IK pour bouger le maillage visuel. On utilise l'IK pour calculer la **pose cible**, puis on utilise des Moteurs Physiques (`JointMotors`) sur le Ragdoll pour qu'il essaie d'atteindre cette pose en utilisant de la force physique.
>
> Cela permet au personnage de trébucher, d'être poussé, tout en essayant de se relever.

5. Exemple : La jambe d'araignée

Vous pouvez regarder cette vidéo:

https://www.youtube.com/watch?v=Ihp6tOCYHug&t=393s

L'objectif est de coder une patte à 3 segments.

- Utiliser un **Raycast** Rapier pour trouver le sol.
- Coder l'algorithme **FABRIK** (ou analytique 2 os) pour calculer la position des jointures.
- Mettre à jour les positions des Meshes Three.js (Kinematic) pour visualiser la patte.

Physique du Jeu : Les 3 Lois de Newton

Ou « Comment programmer l'univers »

Isaac Newton n'a jamais codé en JavaScript, mais sans lui, aucun jeu vidéo moderne n'existerait. Ses trois lois sont les règles fondamentales du moteur physique que nous allons construire.

Les lois de Newton sont des lois mathématiques qui décrivent comment les objets se déplacent et se comportent sous l'effet des forces. Elles nous permettent de donner un comportement réaliste à nos objets dans le jeu.

En physique, on crée des modèles, en se basant sur nos observations, puis on confronte ces modèles à la réalité, pour valider ou invalider ces modèles. À aucun moment les physiciens prétentent que leurs modèles sont exactes, mais une loi est utile à partir du moment ou aucune expérience ne peut prouver qu'elle est fausse.

1. La Première Loi : L'Inertie

Le principe du statu quo

> **La Loi :** Un objet au repos reste au repos, et un objet en mouvement continue en ligne droite à vitesse constante, **sauf** si une force extérieure agit sur lui.

Ce que ça veut dire : Les objets ont une résistance naturelle au changement. Ils veulent continuer à faire ce qu'ils font déjà. Pour changer la vitesse d'un objet (accélérer, freiner, tourner), il **faut** appliquer une force.

Exemple « Vie de tous les jours » : Vous êtes debout dans le bus. Le bus freine brusquement. Votre corps continue d'avancer et vous manquez de tomber. C'est l'inertie : votre corps voulait garder sa vitesse.

Exemple « Jeu Vidéo » :

- **Space Invaders / Mario (Sol)** : Quand vous lâchez la manette, le personnage s'arrête instantanément. *C'est physiquement faux !* (C'est comme s'il y avait des frottements infinis).
- **Asteroids / Dead Space (Espace)** : Vous donnez une impulsion au vaisseau. Même si vous lâchez les commandes, le vaisseau continue d'avancer pour toujours dans la même direction. C'est ça, la vraie inertie (pas de frottements dans l'espace).
- **Niveaux de glace** : Pourquoi Mario glisse ? Parce que les frottements (la force extérieure qui freine) sont réduits. L'inertie reprend le dessus.

---

2. La Deuxième Loi : La Dynamique

L'équation du Moteur Physique

C'est la loi la plus importante pour nous, car c'est celle qu'on code littéralement dans `animate()`.

La Formule : $\vec{F} = m \cdot \vec{a}$

Dans notre cas, on cherche l'accélération :

$$\vec{a} = \frac{\vec{F}}{m}$$

Ce que ça veut dire : L'accélération ($\vec{a}$) dépend de deux choses :
1. Une force : ($\vec{F}$).
2. La lourdeur intrinsèque de l'objet (m).

Exemple « Vie de tous les jours » : Imaginez un caddie de supermarché.
- **Vide (m petit)** : Une petite force le fait partir à toute vitesse.
- **Plein d'eau (m grand)** : La même force le fait à peine bouger.

Exemple « Jeu Vidéo » (Balancing) : Imaginez un jeu de tir (FPS) avec deux classes de personnages :
- **Le Scout (Masse = 50kg)** : Si le joueur appuie sur « Avancer » (Force = 500N), il accélère à 10m/s^2. Il est agile.
- **Le Tank (Masse = 200kg)** : Avec la même touche « Avancer » (Force = 500N), il accélère seulement à 2.5m/s^2. Il est lent et lourd.

Dans Three.js : C'est ici qu'on calcule `acceleration`. Si on veut simuler du vent, de la gravité ou des ressorts, on additionne toutes les forces, on divise par la masse, et on obtient l'accélération pour mettre à jour la vitesse en utilisant la méthode d'Euler. Et de la vitesse, on modifie la position, toujours en utilisant la méthode d'Euler une deuxième fois.

---

3. La Troisième Loi : Action-Réaction

Les forces sont des paires

La Loi : Pour toute action (force), il y a une réaction égale et opposée.

$$\vec{F}_{A \to B} = -\vec{F}_{B \to A}$$

Ce que ça veut dire : On ne peut pas toucher sans être touché. Si vous poussez un mur, le mur vous pousse en retour. Les forces vont toujours par paires.

Exemple « Vie de tous les jours » :
- **Le Skateboard :** Pour avancer, vous poussez le sol vers l'**arrière** avec votre pied. En réaction, le sol pousse votre pied (et vous) vers l'**avant**.
- **Le Ballon de baudruche :** L'air est éjecté vers l'arrière, le ballon part vers l'avant.

Exemple « Jeu Vidéo » :

- **Recul des armes (Recoil)** : Dans *Call of Duty*, quand vous tirez une balle vers l'avant (Action), votre arme et votre caméra remontent vers l'arrière (Réaction).
- **Rocket Jump (Quake / TF2)** : Vous tirez une roquette sur le sol. L'explosion exerce une force vers le bas sur le sol. En réaction, le sol (et l'explosion) exerce une force vers le haut sur le joueur, le propulsant dans les airs.
- **Collisions (Billard)** : Quand la boule blanche tape une boule rouge, la blanche s'arrête (ou recule) car la rouge lui a « rendu » le choc.

Résumé pour le développeur

Loi	Concept	Implémentation Code
1. Inertie	Les objets gardent leur vitesse.	Ne remettez pas `velocity` à 0 à chaque frame ! Laissez-la vivre entre les frames.
2. F = ma	Force → Accélération.	`acc = forces.divideScalar(mass)`
3. Action-Réaction	Les interactions sont doubles.	Collision : Si A repousse B, alors B doit repousser A avec la même force.

Tableau 1. – Aide-mémoire des lois de Newton

Annexe : Pourquoi des objets de masses différentes tombent-ils avec la même vitesse ?

La « force de pesanteur » est la force d'attraction gravitationnelle exercée par un astre (comme la Terre) sur un corps, et qui est aussi appelée « poids ». Elle est calculée par la formule :

$$\vec{P}_g = m.\vec{g}$$

La deuxième loi de Newton nous dit que l'accélération est proportionnelle à la force :

$$\vec{a} = \frac{\vec{F}}{m} = \frac{m.\vec{g}}{m} = \vec{g}$$

Donc, l'accélération est la même pour tous les objets, même si la force qui s'applique à eux est proportionnelle à leur masse.

Annexe : Intégration Numérique & Série de Taylor

Comment l'ordinateur prédit le futur

Dans nos simulations (Three.js), le temps n'est pas continu. L'ordinateur calcule le monde image par image (environ 60 fois par seconde). Nous devons donc transformer les équations physiques continues (dérivées) en instructions discrètes (additions).

1. La Méthode d'Euler

La méthode d'Euler est l'approche la plus intuitive : elle suppose que la vitesse est constante entre deux images.

L'Algorithme (La boucle de jeu)

Pour chaque pas de temps Δt (ex: 0.016s) :

1. **Calculer l'accélération** (Forces / Masse) :
$$\vec{a}_n = \frac{\vec{F}}{m}$$
2. **Mettre à jour la vitesse** (Vitesse actuelle + Changement) :
$$\vec{v}_{n+1} = \vec{v}_n + \vec{a}_n \cdot \Delta t$$
3. **Mettre à jour la position** (Position actuelle + Déplacement) :
$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n \cdot \Delta t$$

2. D'où ça vient ? (La Dérivée Discrète)

Mathématiquement, la vitesse est la dérivée de la position. Si on enlève la limite $\lim_{\Delta t \to 0}$, on obtient une approximation :

$$\vec{v}(t) = \frac{d\vec{r}}{dt} \approx \frac{\vec{r}(t + \Delta t) - \vec{r}(t)}{\Delta t}$$

En isolant le terme futur $\vec{r}(t + \Delta t)$, on retombe sur la formule d'Euler :

$$\underbrace{\vec{r}(t + \Delta t)}_{\text{Futur}} \approx \underbrace{\vec{r}(t)}_{\text{Présent}} + \underbrace{\vec{v}(t) \cdot \Delta t}_{\text{Pas}}$$

3. La Série de Taylor (Le Moteur Mathématique)

La méthode d'Euler n'est en fait que la « pointe de l'iceberg » d'un concept plus puissant : la **Série de Taylor**. Elle stipule que n'importe quelle fonction (trajectoire) peut être reconstruite en additionnant ses dérivées successives.

L'intuition du conducteur aveugle : Si vous fermez les yeux au volant, comment deviner où vous serez dans 1 seconde (Δt) ?

- **Ordre 0 (Position)** : « Je ne bouge pas. » $\rightarrow \vec{r}(t)$
- **Ordre 1 (Vitesse)** : « J'avance tout droit. » $\rightarrow \vec{v}(t)\Delta t$

- **Ordre 2 (Accélération)** : « J'accélère, donc je courbe. » $\rightarrow \frac{1}{2}\vec{a}(t)\Delta t^2$
- **Ordre 3 (A-coup)** : « J'appuie de plus en plus fort... » $\rightarrow \dots$

La Formule Complète :

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2 + \frac{1}{6}\vec{j}(t)\Delta t^3 + \dots$$

Pourquoi Euler est une approximation ?

La méthode d'Euler **coupe** la série de Taylor après le terme de vitesse.

$$\begin{aligned}\vec{r}_{n+1} &= \vec{r}_n + \vec{v}_n\Delta t \\ &+ \underbrace{O(\Delta t^2)}_{\text{Termes ignorés (Erreur)}}\end{aligned}$$

C'est pour cela qu'on dit qu'Euler est une méthode du **Premier Ordre**. L'erreur commise à chaque pas est proportionnelle au carré du pas de temps (Δt^2).

4. Conséquences Pratiques

Puisque Euler ignore la courbure de la trajectoire (l'accélération) à l'intérieur d'un pas de temps, il a tendance à « déraper » vers l'extérieur des virages ou à gagner de l'énergie (spirale vers l'extérieur) dans des systèmes orbitaux.

Euler (Ordre 1)	**Intégration exacte (Ordre 2)**
$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n\Delta t$ *Prend la pente au début et trace une ligne droite.*	$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n\Delta t + \frac{1}{2}\vec{a}\Delta t^2$ *Prend en compte la courbure (accélération).*
Facile à coder. Rapide.	Exact pour la gravité constante (projectiles).

Tableau 2. – Comparaison pour un pas de temps Δt

Conclusion pour le projet : Pour des projectiles simples soumis à une gravité constante, nous pourrions utiliser la formule exacte (Ordre 2). Cependant, Euler reste très utile car il fonctionne même quand l'accélération change tout le temps (ex: vent, ressorts, collisions), là où la formule exacte devient trop complexe.

Annexe Mathématique : Dérivée et Intégration 101

Pour faire de la physique, nous avons besoin de deux outils mathématiques fondamentaux qui fonctionnent comme des opérations inverses l'une de l'autre, un peu comme l'addition et la soustraction.

1. La Dérivée (La Pente)

Le microscope du changement

La dérivée mesure comment une valeur change **à un instant précis**. Géométriquement, c'est la **pente** de la courbe.

- **Notation :** Si on a une fonction $f(t)$, sa dérivée s'écrit $f'(t)$ ou $\frac{df}{dt}$.
- **Intuition :**
  - Si $x(t)$ est la position, sa pente nous dit à quelle vitesse on avance. Une pente raide = grande vitesse. Une pente nulle (plat) = à l'arrêt.
  - $v(t) = \frac{dx(t)}{dt}$

Règle de puissance (La plus utile !) : Pour dériver t^n, on descend l'exposant devant et on réduit l'exposant de 1.

$$f(t) = t^n \longrightarrow f'(t) = nt^{n-1}$$

Exemple : Si $x(t) = 5t^3$, alors $v(t) = 15t^2$.

2. L'Intégrale (L'Aire)

L'accumulateur

L'intégration est l'opération inverse de la dérivée. Elle permet de retrouver la fonction originale à partir de son taux de changement. Géométriquement, c'est l'**aire sous la courbe**.

- **Notation :** $\int f(t)dt$.
- **Intuition :**
  - Si on connaît la vitesse à chaque instant, l'intégrale additionne toutes ces petites distances parcourues pour nous donner la position totale.
  - $x(t) = \int v(t)dt$

Règle de puissance inversée : On augmente l'exposant de 1 et on divise par le nouveau nombre.

$$f(t) = t^n \longrightarrow F(t) = \frac{t^{n+1}}{n+1} + C$$

Attention à la Constante (C) ! Quand on intègre, on perd l'information du point de départ. La constante C représente les conditions initiales (Position initiale x_0 ou Vitesse initiale v_0).

3. Le cycle de la Cinématique

En physique, on passe constamment de l'un à l'autre selon ce que l'on cherche :

Position $x(t)$ $\xrightarrow[\text{Dérivation (Pente)}]{\frac{\vec{d}}{dt}}$ **Vitesse** $v(t)$

Vitesse $v(t)$ $\xrightarrow[\text{Dérivation (Pente)}]{\frac{\vec{d}}{dt}}$ **Accélération** $a(t)$

Et dans l'autre sens…

Accélération $a(t)$ $\xrightarrow[\text{Intégration (Aire + } v_0\text{)}]{\overrightarrow{\int_{t_0}^{t} dt}}$ **Vitesse** $v(t)$

Vitesse $v(t)$ $\xrightarrow[\text{Intégration (Aire + } x_0\text{)}]{\overrightarrow{\int_{t_0}^{t} dt}}$ **Position** $x(t)$

Aide-mémoire des fonctions usuelles

Voici les transformations que nous utiliserons le plus souvent dans ce cours :

Fonction $f(t)$	**Dérivée** $f'(t)$	**Intégrale** $\int f(t)dt$
Constante k (ex: 5)	0	$kt + C$
Linéaire t	1	$\frac{1}{2}t^2 + C$
Quadratique t^2	$2t$	$\frac{1}{3}t^3 + C$
Sinus $\sin(\omega t)$	$\omega \cos(\omega t)$	$-\frac{1}{\omega}\cos(\omega t) + C$
Cosinus $\cos(\omega t)$	$-\omega \sin(\omega t)$	$\frac{1}{\omega}\sin(\omega t) + C$

Tableau 3. – Tableau des dérivées et primitives usuelles

TP Avancé : La Méthode Runge-Kutta (RK4)

Le « Couteau Suisse » de la simulation physique

Dans les sessions précédentes, nous avons utilisé la méthode d'Euler. Elle est simple, mais elle « dérape » vite car elle suppose que la vitesse est constante durant tout le pas de temps.

La méthode de Runge-Kutta d'ordre 4 (RK4) corrige ce problème en étant beaucoup plus astucieuse : au lieu de regarder la pente (dérivée) uniquement au début, elle « sonde » le terrain à 4 endroits différents pour trouver une trajectoire moyenne quasi-parfaite.

1. Le Concept : 4 sondes valent mieux qu'une

Imaginez que vous êtes aveugle et que vous devez traverser une vallée vallonnée en un pas de temps Δt.

- **Euler (k1)** : Vous tendez le pied, sentez la pente actuelle, et faites un grand saut dans cette direction. $\rightarrow$ *Risqué.*
- **RK4** : Vous envoyez des éclaireurs virtuels avant de bouger.

1. **k1 (Le Départ)** : Pente au début (comme Euler).
2. **k2 (Le Milieu A)** : On utilise la pente $k1$ pour avancer jusqu'à la moitié du pas ($\Delta \frac{t}{2}$) et on regarde la pente là-bas.
3. **k3 (Le Milieu B)** : On se méfie de $k2$. On repart du début, mais cette fois on utilise la pente $k2$ pour aller au milieu. On regarde la nouvelle pente.
4. **k4 (L'Arrivée)** : On utilise la pente $k3$ pour aller jusqu'à la fin du pas (Δt) et on regarde la pente finale.

Le Grand Final : On fait une moyenne pondérée de ces 4 pentes. On donne plus de poids aux pentes du milieu ($k2$ et $k3$).

$$\text{Pente Finale} = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

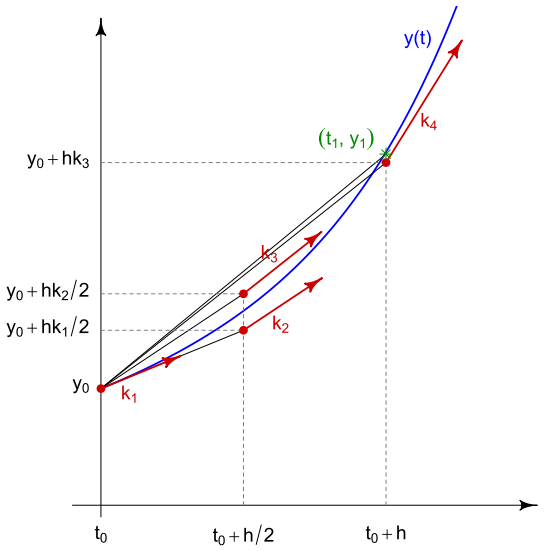


Fig. 17. – Runge Kutta : 4 pentes pour une meilleure trajectoire

2. Les Mathématiques (L'État du Système)

Pour implémenter RK4 proprement, nous devons regrouper nos variables (Position et Vitesse) dans un seul vecteur que nous appellerons l'État (Y).

Si on a un système physique simple :

$$Y = \begin{pmatrix} x \\ v \end{pmatrix}$$

$$\dot{Y} = f(t, Y) = \begin{pmatrix} v \\ a \end{pmatrix}$$

La dérivée de l'état (le changement), c'est la vitesse et l'accélération.

Voici l'algorithme complet pour un pas de temps h (ou Δt) :

Algorithme RK4 pour un pas h :

Soit la fonction `eval(état)` qui retourne la dérivée `[vitesse, accélération]`.

1. k_1 **(Début)** = `eval(état_actuel)`
2. k_2 **(Milieu)** = `eval(état_actuel` $+ k_1 \times \frac{h}{2}$`)`
3. k_3 **(Milieu)** = `eval(état_actuel` $+ k_2 \times \frac{h}{2}$`)`
4. k_4 **(Fin)** = `eval(état_actuel` $+ k_3 \times h$`)`

Nouvel État = `état_actuel` $+ \frac{k_1+2k_2+2k_3+k_4}{6} \times h$

3. Implémentation (Pseudo-Code / JavaScript)

Contrairement à Euler où l'on écrit `pos += vel * dt` directement, RK4 nécessite une structure plus organisée.

```
// Définition d'un État
struct State {
    Vector3 position;
    Vector3 velocity;
};

// La dérivée de l'état, c'est ce qui change
struct Derivative {
    Vector3 d_position; // C'est la vitesse
    Vector3 d_velocity; // C'est l'accélération (Forces / Masse)
};

// Fonction qui calcule les forces et retourne la dérivée
function evaluate(initialState, t, dt, derivative) {
    // 1. Estimer l'état futur basé sur la dérivée précédente
    State state = initialState;
    state.position += derivative.d_position * dt;
    state.velocity += derivative.d_velocity * dt;

    // 2. Calculer les forces à cet endroit/vitesse là
```

```
    // (Exemple: Gravité + Vent + Ressort)
    Vector3 forces = calculateForces(state.position, state.velocity);
    Vector3 acceleration = forces / mass;

    // 3. Retourner la nouvelle dérivée
    return new Derivative(state.velocity, acceleration);
}

// BOUCLE PRINCIPALE (INTÉGRATEUR)
function integrateRK4(state, t, dt) {
    // a. Préparer les 4 échantillons
    Derivative a = evaluate(state, t, 0.0, new Derivative()); // k1
    Derivative b = evaluate(state, t, dt*0.5, a);             // k2
    Derivative c = evaluate(state, t, dt*0.5, b);             // k3
    Derivative d = evaluate(state, t, dt, c);                 // k4

    // b. Moyenne pondérée pour la position (dxdt)
    Vector3 dxdt = (a.d_pos + 2*(b.d_pos + c.d_pos) + d.d_pos) * 1/6;

    // c. Moyenne pondérée pour la vitesse (dvdt)
    Vector3 dvdt = (a.d_vel + 2*(b.d_vel + c.d_vel) + d.d_vel) * 1/6;

    // d. Mise à jour finale
    state.position += dxdt * dt;
    state.velocity += dvdt * dt;
}
```

Session 9 : L'Intégration de Verlet

Qu'est-ce que l'Intégration de Verlet ?

Contrairement à l'intégration d'Euler, l'algorithme de Verlet ne stocke pas explicitement la vitesse d'un objet.

La vitesse est déduite de la différence entre la position actuelle et la position à l'instant précédent. C'est un schéma d'intégration symplectique, ce qui signifie qu'il conserve l'énergie de manière bien plus stable qu'Euler.

Le Concept de Base

Imaginez que vous savez où vous êtes (P_n) et où vous étiez il y a un instant ($P_{\{n-1\}}$). La direction et la distance entre ces deux points représentent votre élan (inertie).

La Formule Mathématique

La position future ($x_{\{n+1\}}$) est calculée à partir de la position actuelle (x_n) et de la précédente ($x_{\{n-1\}}$) :

$$x_{\{n+1\}} = x_n + \left(x_n - x_{\{n-1\}}\right) + a \cdot dt^2$$

Où :
- $\left(x_n - x_{\{n-1\}}\right)$ représente la vitesse (pseudo-vitesse).
- $a \cdot dt^2$ représente l'accélération (comme la gravité).

L'Algorithme en 3 Étapes

Pour chaque objet dans la simulation, on suit cet ordre strict :

1. Calcul de la Vitesse : `vitesse = pos - old_pos`
2. Mise à jour :
   - On sauve la position actuelle : `temp = pos`
   - On calcule la nouvelle : `pos = pos + vitesse + (accel * dt * dt)`
   - On met à jour l'ancienne : `old_pos = temp`
3. Contraintes : On ajuste `pos` directement (murs, collisions).

Comparaison : Euler vs Verlet.

Euler Explicite :
- Avantage : Très simple.

- Inconvénient : « Explose » vite. Si on modifie la position d'un objet sans changer sa vitesse, le moteur devient incohérent.

Verlet :

- Avantage : **Inconditionnellement stable**. Si on déplace un objet manuellement (contrainte), sa vitesse s'ajuste d'elle-même à la frame suivante.
- Idéal pour : Les tas de billes, les cordes, les tissus et les chevelures.

Gestion des Collisions

En Verlet, on ne calcule pas de forces de réaction complexes. On utilise la Relaxation de Contrainte.

Si une bille pénètre dans un mur :

- On la téléporte à la surface du mur (`pos.y = sol`).
- À la prochaine frame, le calcul `pos - old_pos` donnera une vitesse nulle ou réduite.
- Le système s'auto-équilibre.

$$v_{\text{new}} = \frac{\text{pos}_{\text{corrigée}} - \text{old}_{\text{pos}}}{\text{dt}}$$

L'Impulsion de Collision

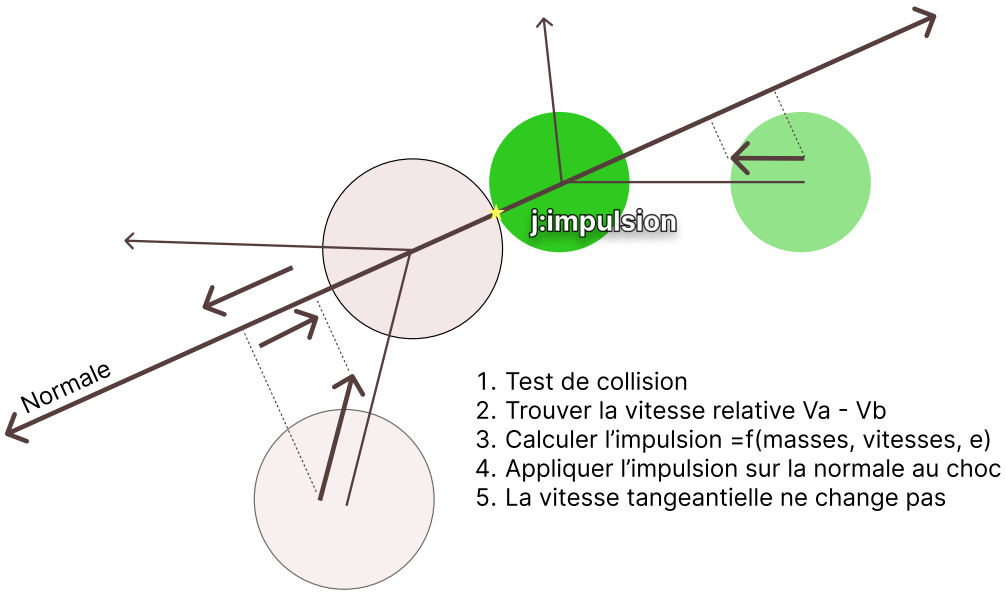


Fig. 18. – Choc entre deux objets

Partie 1 : D'où vient l'Impulsion ? (Le Lien Newtonien)

La conservation de la quantité de mouvement est une loi fondamentale de la physique, et elle nous permet de calculer facilement les changements de vitesse de 2 objets en collision.

1. Retour à la définition de la Force L'accélération $\vec{a}$ est la dérivée de la vitesse.

$$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt}$$

Comme la masse est constante, on peut la rentrer dans la dérivée. On reconnait alors la Quantité de Mouvement ($\vec{p} = m\vec{v}$).

$$\vec{F} = \frac{d(m\vec{v})}{dt} = \frac{d\vec{p}}{dt}$$

2. L'Intégration (Somme des forces) Pour connaitre l'effet total d'un choc entre un instant t_1 et t_2, on multiplie par dt et on intègre (on somme) les deux côtés de l'équation :

$$\int_{t_1}^{t_2} \vec{F} dt = \int_{p_1}^{p_2} d\vec{p}$$

3. Le Résultat : Le Théorème de l'Impulsion

- Le terme de droite devient simplement le changement de quantité de mouvement : $\Delta\vec{p}$.
- Le terme de gauche est la définition exacte de l'Impulsion $\vec{J}$.

$$\vec{J} = \Delta\vec{p}$$

Interprétation Géométrique : L'impulsion est l'**Aire sous la courbe** de la Force en fonction du temps. Peu importe que la force soit petite et longue, ou géante et courte : si l'aire est la même, le changement de vitesse à l'instant t_2 est le même.

Dans une simulation « Narrow Phase », on considère que la collision est instantanée ($\Delta t \approx 0$).

Si le temps est nul, pour avoir une aire non-nulle, la Force devrait être infinie. L'ordinateur ne peut pas calculer l'infini.

L'Astuce : On saute l'étape de l'intégration temporelle. On utilise directement le résultat J. Cela nous permet de « téléporter » la vitesse sans passer par l'accélération.

$$\Delta v = \frac{J}{m}$$

Partie 2 : La Règle du Rebond (La Loi)

Nous avons l'outil (J), mais quelle valeur doit-il avoir ? C'est ici qu'intervient la **Loi expérimentale**.

Loi de Restitution : « La vitesse à laquelle deux objets s'éloignent après un choc est proportionnelle à la vitesse à laquelle ils se rapprochaient avant le choc. »

$$v_{\text{rel}}(\text{après}) = -e \cdot v_{\text{rel}}(\text{avant})$$

- $v_{\text{rel}} = v_A - v_B$ (Vitesse relative le long de la normale).
- e : Coefficient de restitution (entre 0 et 1).
- Le signe moins ($-$) indique que les objets repartent dans l'autre sens.

Zoom sur la Vitesse Relative

Avant de calculer, définissons les termes. Soient $\vec{v}_A$ et $\vec{v}_B$ les vecteurs vitesse des deux objets dans le monde.

1. Le Vecteur Vitesse Relative ($\Delta\vec{v}$) C'est la vitesse de A vue depuis B.

$$\vec{v}_{\text{diff}} = \vec{v}_A - \vec{v}_B$$

2. La Normale de Collision ($\vec{n}$) C'est le vecteur unitaire qui relie les centres (la direction du choc).

3. La Vitesse Relative Scalaire (v_{rel}) C'est celle qu'on utilise dans la formule de l'impulsion ! On ne s'intéresse qu'à la vitesse **sur l'axe du choc**. On fait donc une projection (Produit Scalaire) :

$$v_{\text{rel}} = (\vec{v}_A - \vec{v}_B) \cdot \vec{n}$$

Interprétation du signe :
- Si $v_{\text{rel}} < 0$: Les objets se rapprochent (Collision imminente).
- Si $v_{\text{rel}} > 0$: Les objets s'éloignent (Ils se sont déjà tapés ou se fuient).
- Si $v_{\text{rel}} = 0$: Ils glissent l'un à côté de l'autre sans s'écraser.

Application de l'Impulsion

Cherchons l'intensité de l'impulsion j (scalaire) à appliquer le long de la normale pour satisfaire la loi du rebond.

On applique j sur la boule A et $-j$ sur la boule B (3ème loi de Newton). Les nouvelles vitesses (v') sont :

$$\boldsymbol{v'_A = v_A + \frac{j}{m_A}} \quad \textbf{et} \quad \boldsymbol{v'_B = v_B - \frac{j}{m_B}}$$

En passant par les impulsions, on peut effectuer un changement de vitesse instantané, sans passer par l'accélération.

Calcul de la valeur de l'impulsion

Voici la formule de base pour calculer l'impulsion :

$$\boldsymbol{j = \frac{-(1+e)v_{\text{rel}}}{\frac{1}{m_A} + \frac{1}{m_B}}}$$

Le terme au dénominateur $\left(\frac{1}{m_A} + \frac{1}{m_B}\right)$ est peu intuitif. Transformons-le. Mise au même dénominateur :

$$\frac{1}{m_A} + \frac{1}{m_B} = \frac{m_B}{m_A m_B} + \frac{m_A}{m_A m_B} = \frac{m_A + m_B}{m_A m_B}$$

Maintenant, remplaçons cette fraction dans la formule de j. Diviser par une fraction, c'est multiplier par son inverse :

$$j = -(1+e)v_{\text{rel}} \cdot \frac{1}{\frac{m_A+m_B}{m_A m_B}}$$

Ce qui nous donne la **Formule Finale** très élégante :

$$\boldsymbol{j = \underbrace{-(1+e)v_{\text{rel}}}_{\text{Vitesse à changer}} \cdot \underbrace{\left(\frac{m_A m_B}{m_A + m_B}\right)}_{\text{Masse Équivalente}}}$$

Interprétation Physique

Cette forme

$$\frac{m_A m_B}{m_A + m_B}$$

est appelée la **Masse Réduite** du système.

Elle montre que lors d'une collision, le système se comporte comme une masse unique équivalente.

- Si une masse est minuscule ($m_A \ll m_B$), la masse équivalente devient $\approx m_A$.
  Conclusion : C'est le plus léger qui dicte la dynamique du rebond.