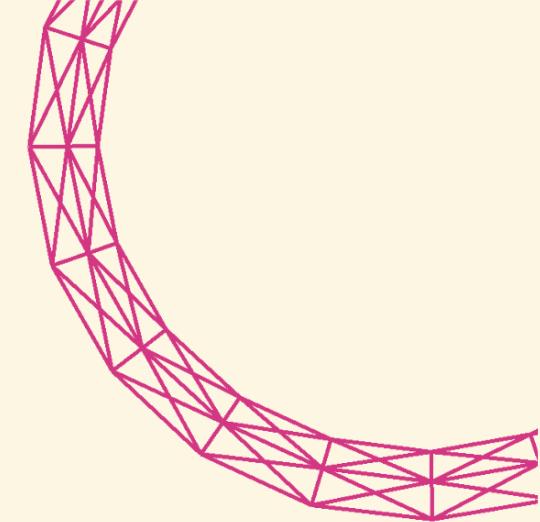
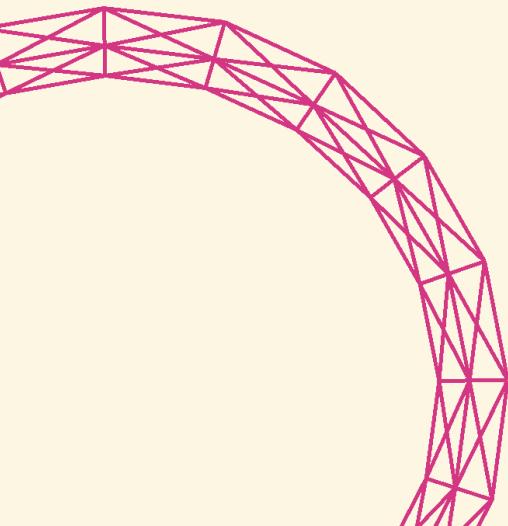


Bartolomeo Ryan 10208
Jeux et Sports



Modélisation de solides déformables



Plan

I. Présentation et première approche

- A. Inspiration
- B. Modélisation mouvement cohérent avec la réalité physique

II. Chute d'un solide déformable

- A. Mise en place d'un système résistant aux perturbations :
 - Méthode d'intégration d'Euler explicite
 - Méthode de Runge et Kutta
- B. Problème de la réaction du support
 - Force électrostatique
 - Méthode de Backtracking

III. Optimisation par assimilation à un gaz

- A. Modèle du gaz parfait
- B. Théorème de Stokes
- C. Expansion en 3 dimensions

Présentation du problème et première approche

Motivation

Wobbledogs (wobbledogs.com)

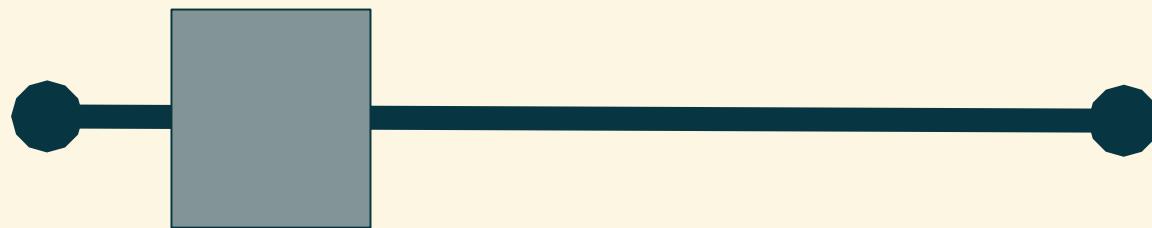


- Multijoueur
- Affichage graphique
- Temps réel
- Comment minimiser le temps de calcul dans la simulation de solides déformable ?

Gang Beast
(<https://www.youtube.com/watch?v=LGo9RpPrw3U>)

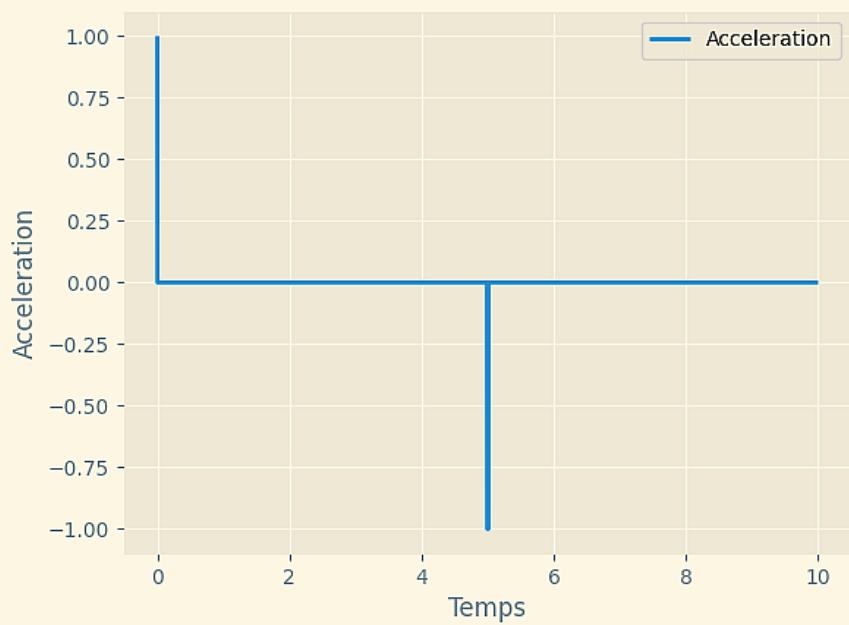
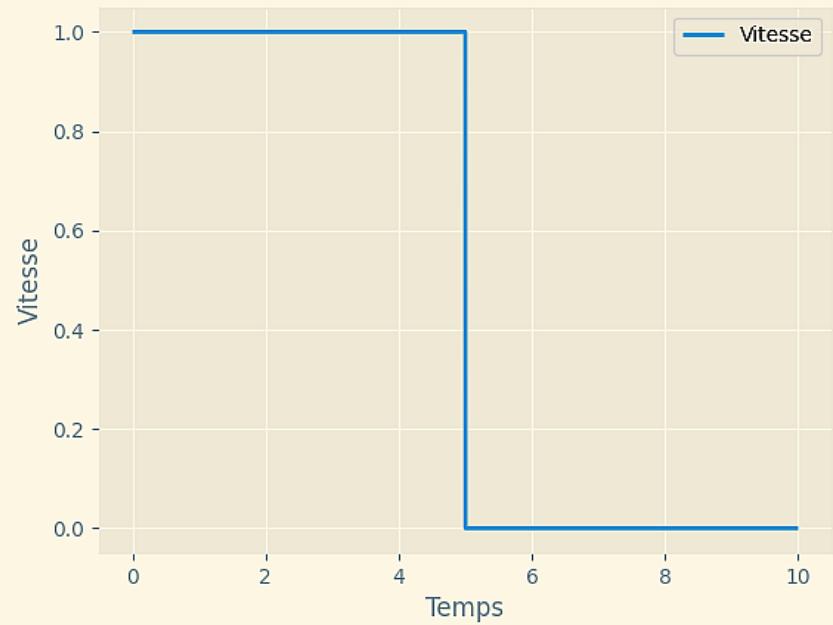
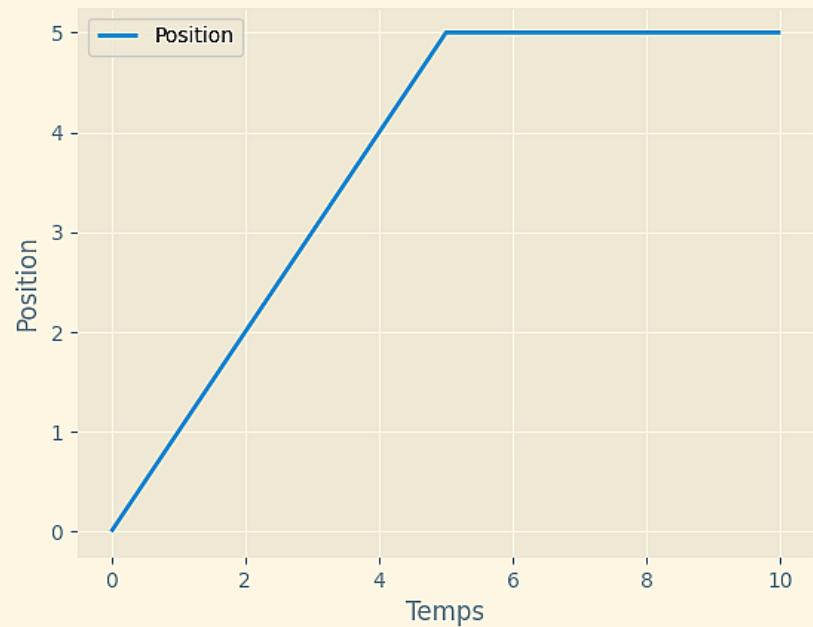
Modélisation réaliste d'un mouvement

Cas simple

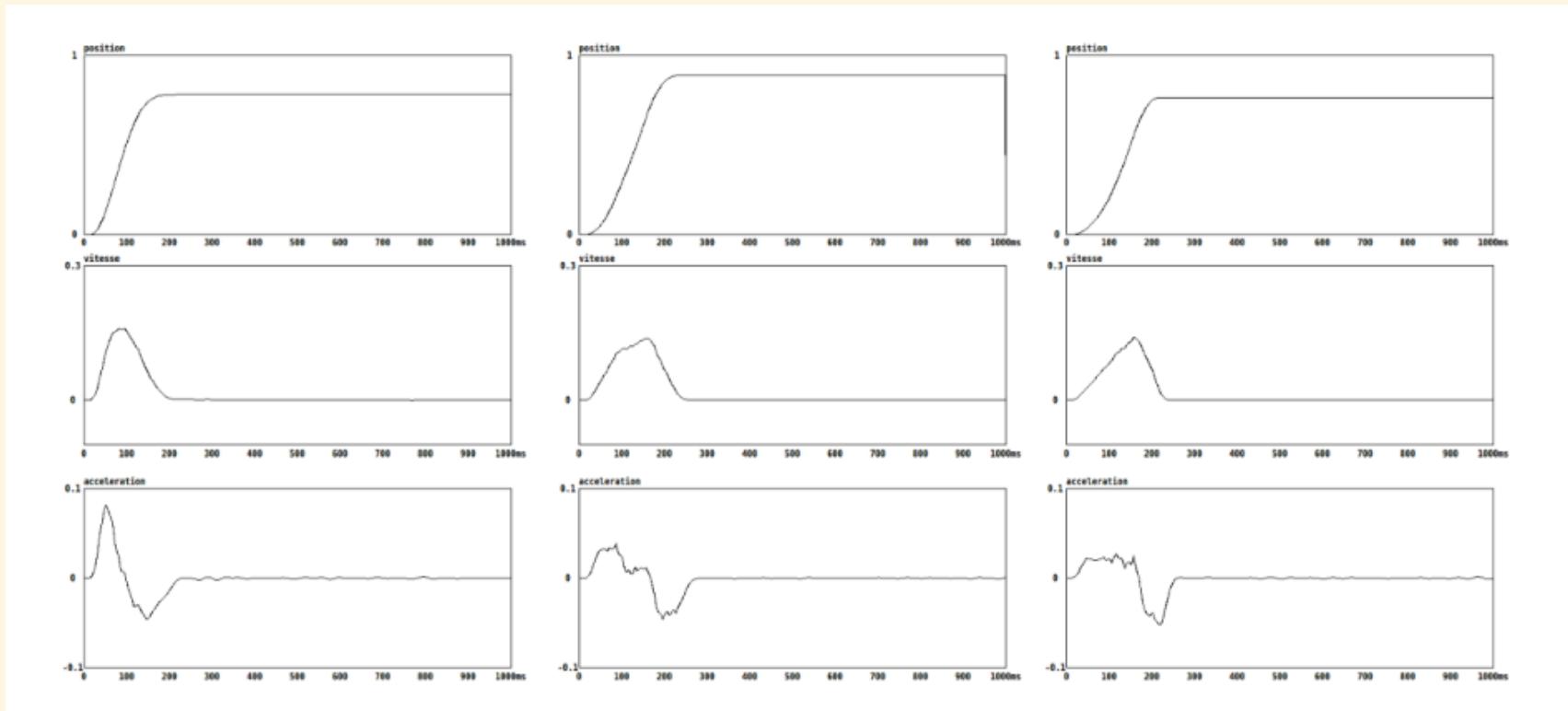


Modèle informatique simple

Profil de déplacement linéaire



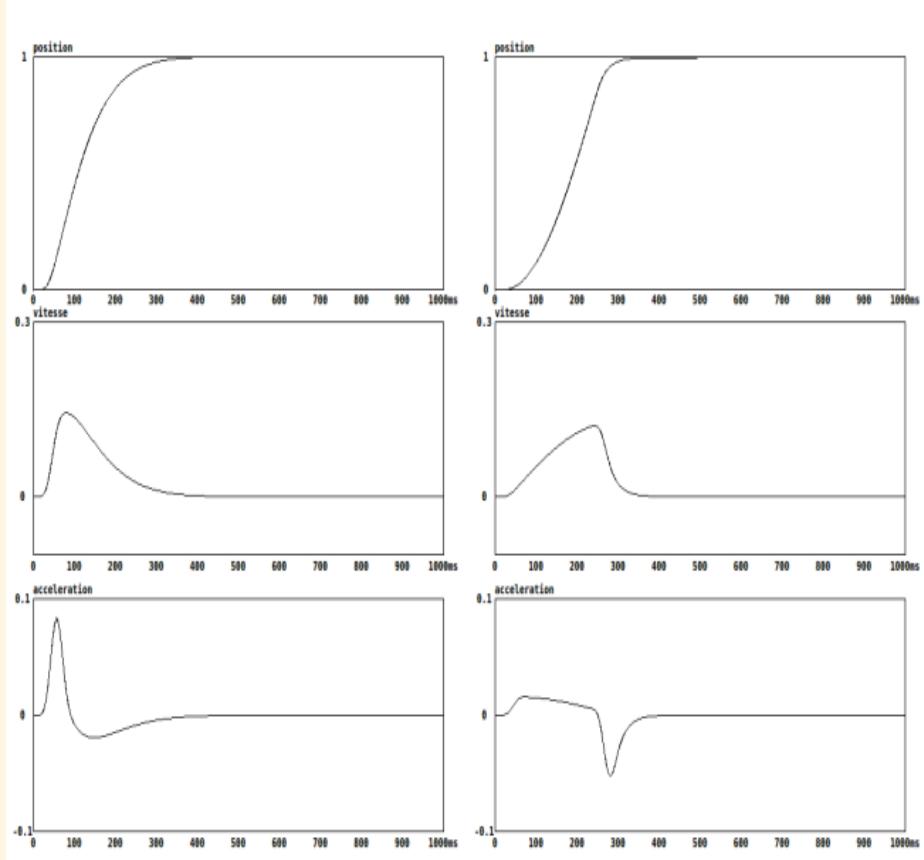
Mouvement naturel



Mass-Spring-System model for real time expressive behaviour synthesis ~ Cyrille Henry

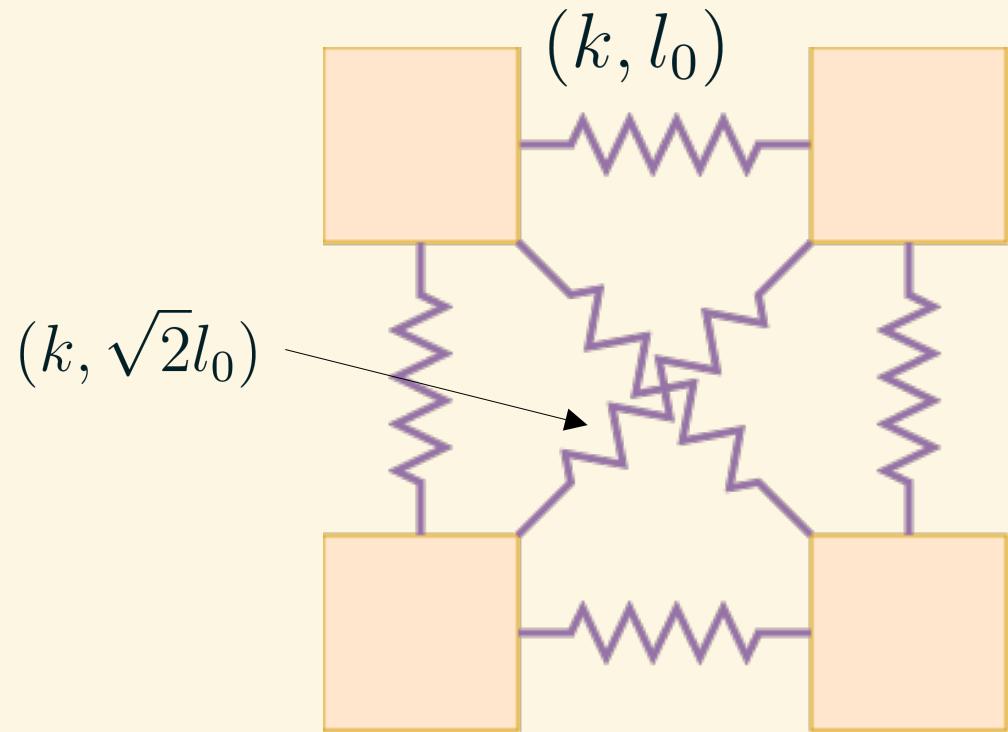
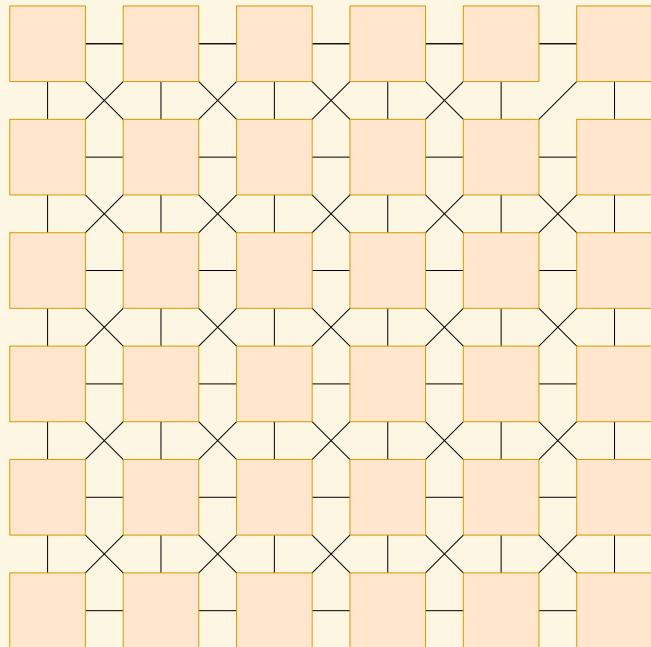
Mouvements type Système masse-ressort

Deux masses en mouvement
l'une par rapport à
l'autre



Mass-Spring-System model for real time expressive behaviour synthesis ~ Cyrille Henry

Discrétisation du solide

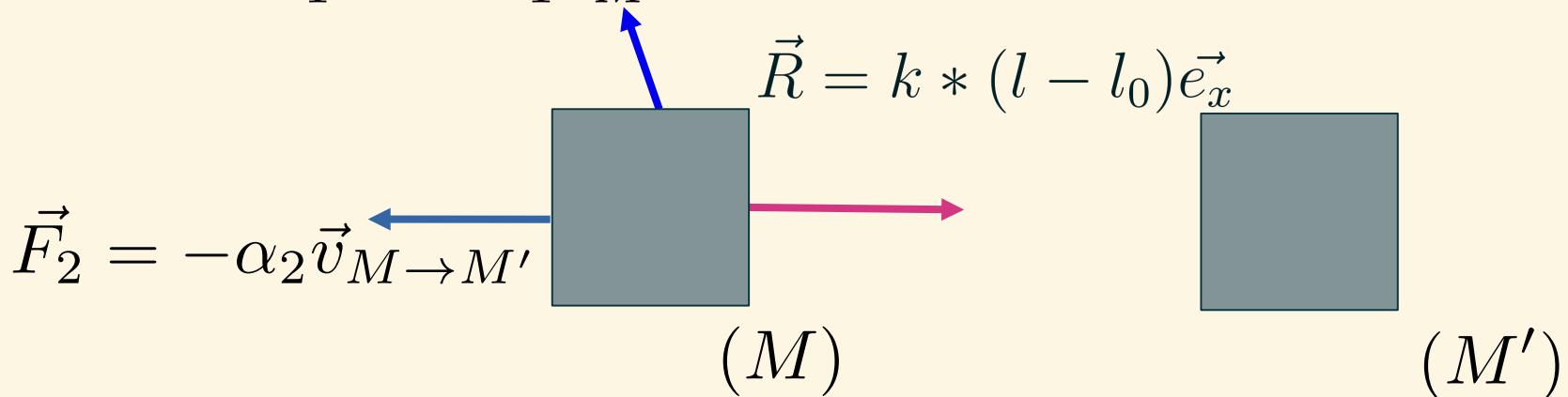


II. Réalisation

1. Système stable malgré une perturbation

Méthode d'Euler explicite

$$\vec{F}_1 = -\alpha_1 \vec{v}_M$$



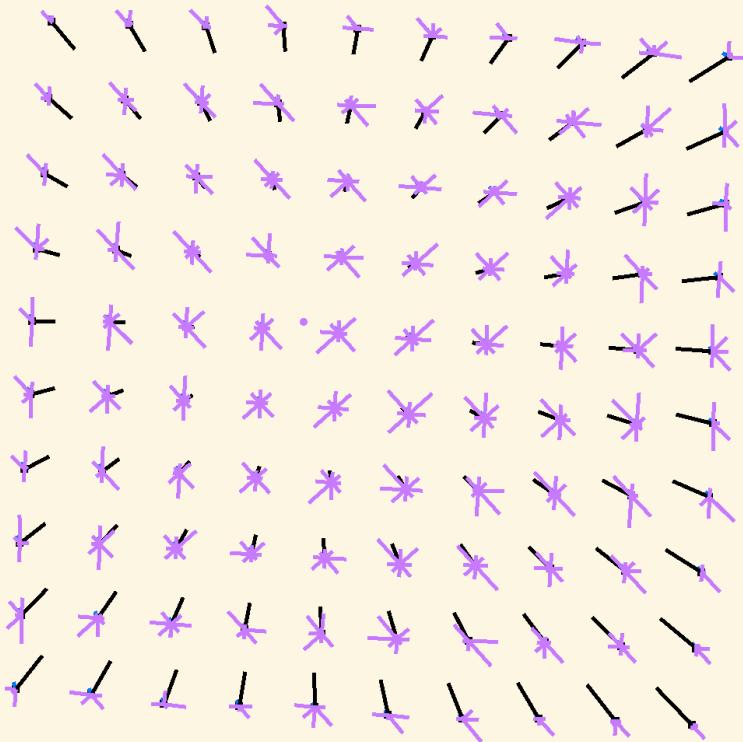
$$x[t + 1] = x[t] + v[t] * dt$$

$$v[t + 1] = v[t] + \text{acceleration}(t) * dt$$

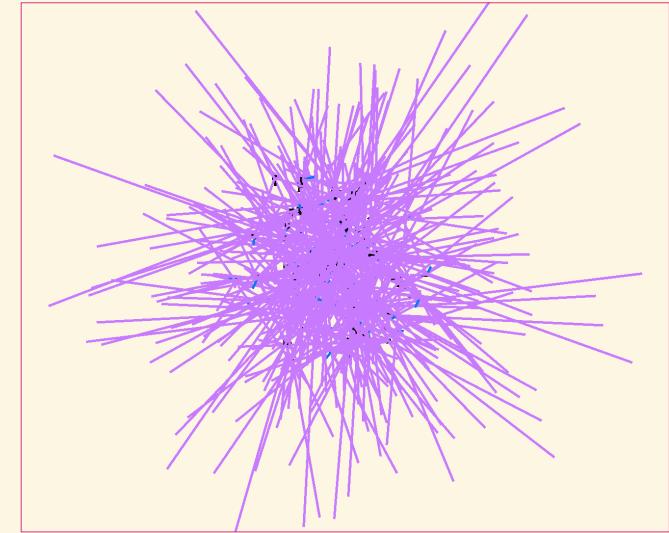
Résultat de la simulation avec la méthode d'Euler

Compression sous l'effet de la force fictive

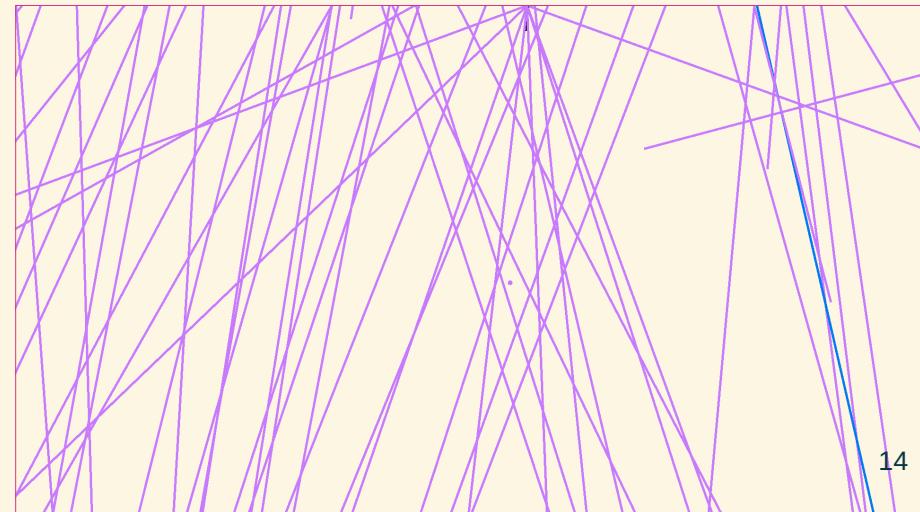
Etat initial



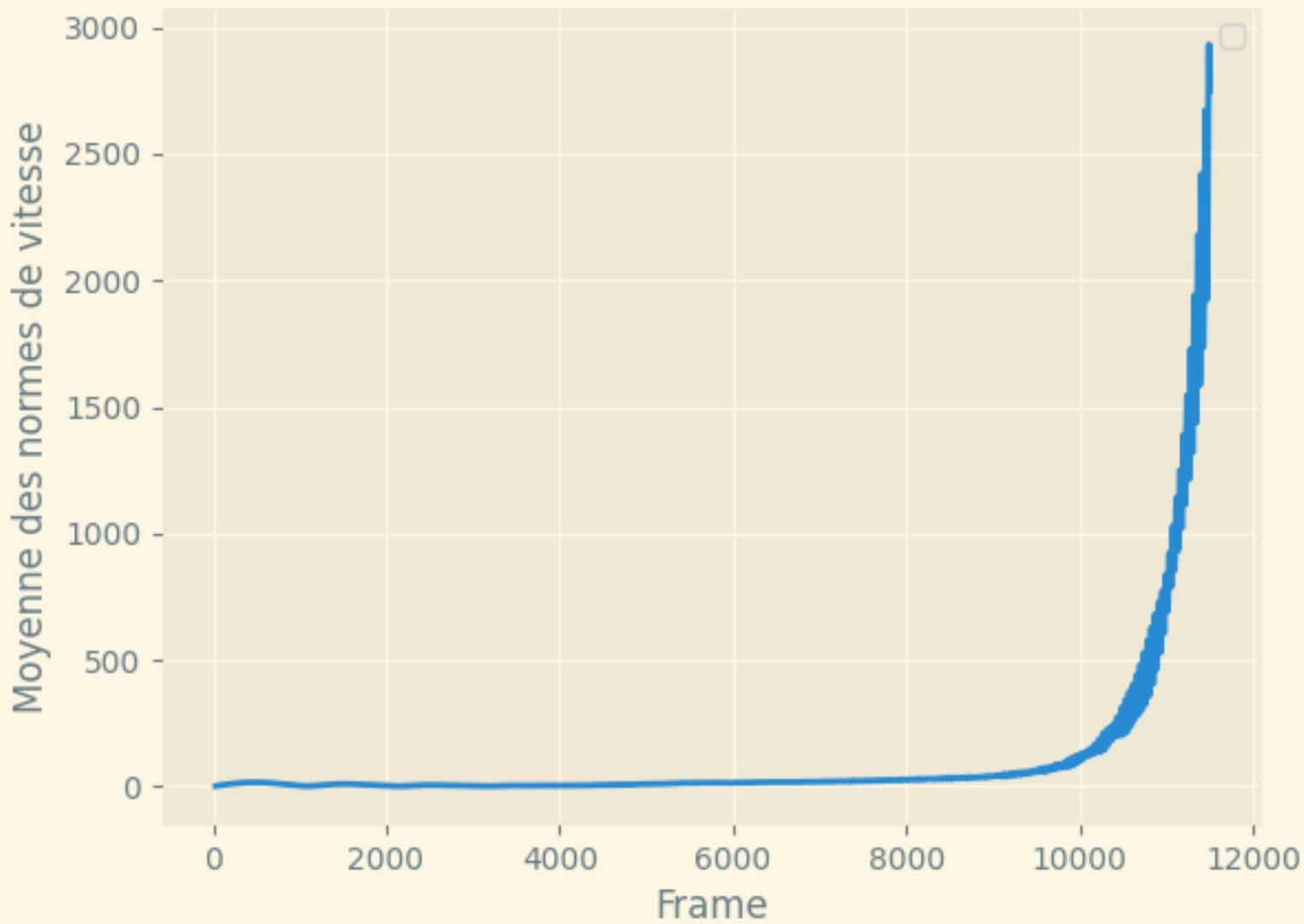
Force élastique en violet
Force fictive en noir



Explosion du système



Premier résultat



Méthode de Runge-Kutta

Approximation de Runge-Kutta

Tableau des coefficients utilisés

Problème de Cauchy à résoudre

$$\begin{cases} y'(t) = f(t, y) \\ y(t_0) = y_0 \end{cases}$$

(c_i)	0	0	0	0	$0(a_{i,j})$
$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	
$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	
1	0	0	1	0	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$	(b_j)

Si z est solution, on a :

$$t_{n,i} = t_n + c_i \times h$$

$$z(t_{n+1}) = z(t_n) + h \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j}))$$

$$z(t_{n,i}) = z(t_n) + h \sum_{j < i} a_{i,j} f(t_{n,j}, z(t_{n,j})))$$

Approximation de Runge-Kutta

Problème différentiel physique

$$\begin{cases} y'' = \text{acc}(t, y, y') \\ y'(t_0) = y'_0 \\ y(t_0) = y_0 \end{cases}$$

On pose alors :

$$Y(t) = \begin{pmatrix} y'(t) \\ y(t) \end{pmatrix}$$

$$Y(t_0) = \begin{pmatrix} y'_0 \\ y_0 \end{pmatrix}$$

$$\text{d'où } Y'(t) = \begin{pmatrix} y''(t) \\ y'(t) \end{pmatrix} = F(t, Y) = \begin{pmatrix} \text{acc}(t, y, y') \\ y'(t) \end{pmatrix}$$

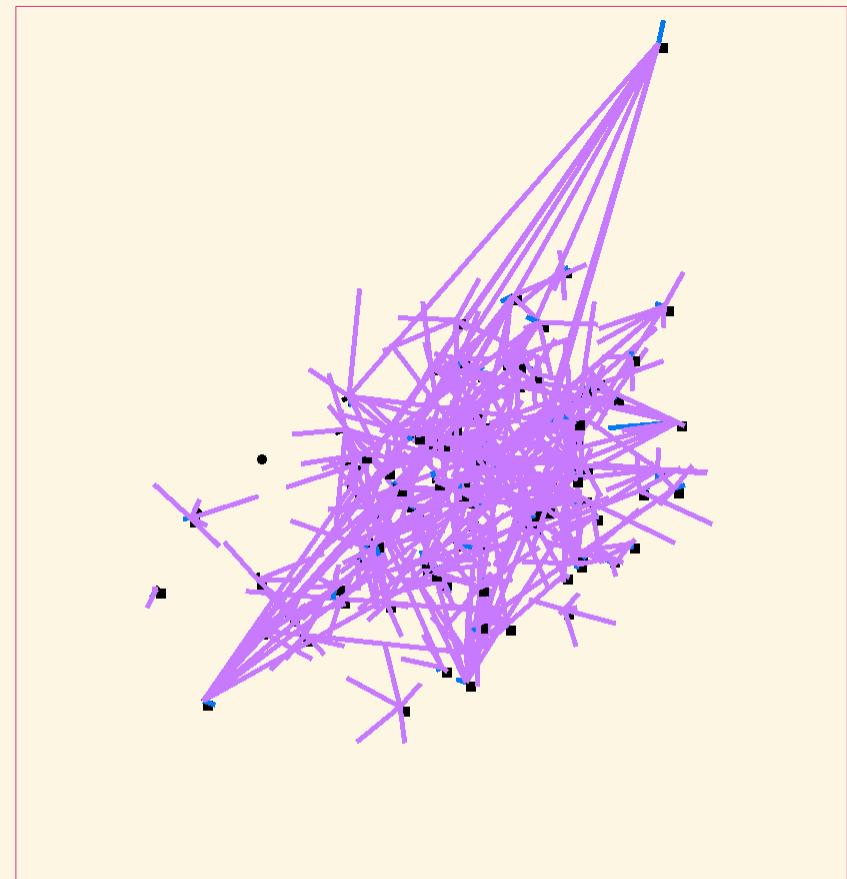
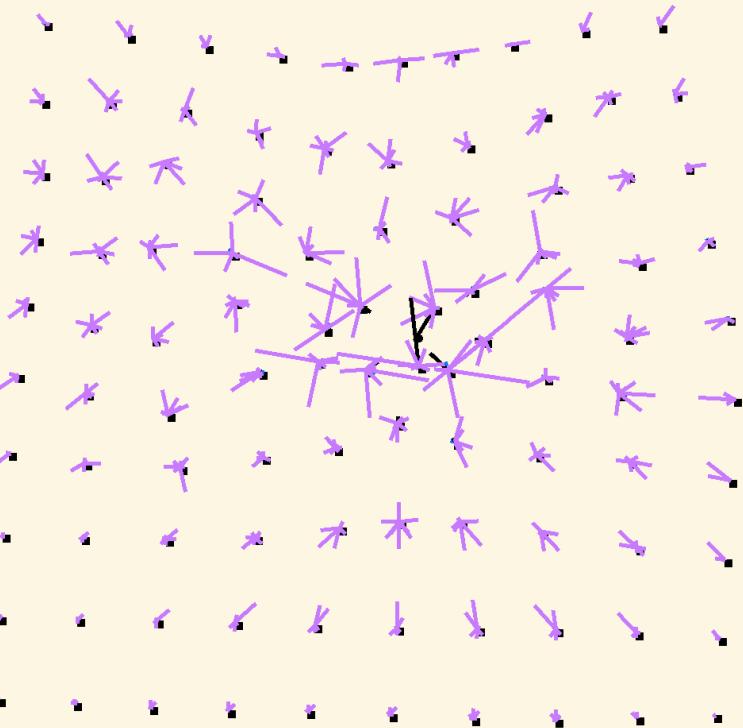
On a alors le problème de Cauchy
d'ordre 1 :

$$\begin{cases} Y'(t) = F(t, Y) \\ Y(t_0) = Y_0 \end{cases}$$

Résultat d'algorithme

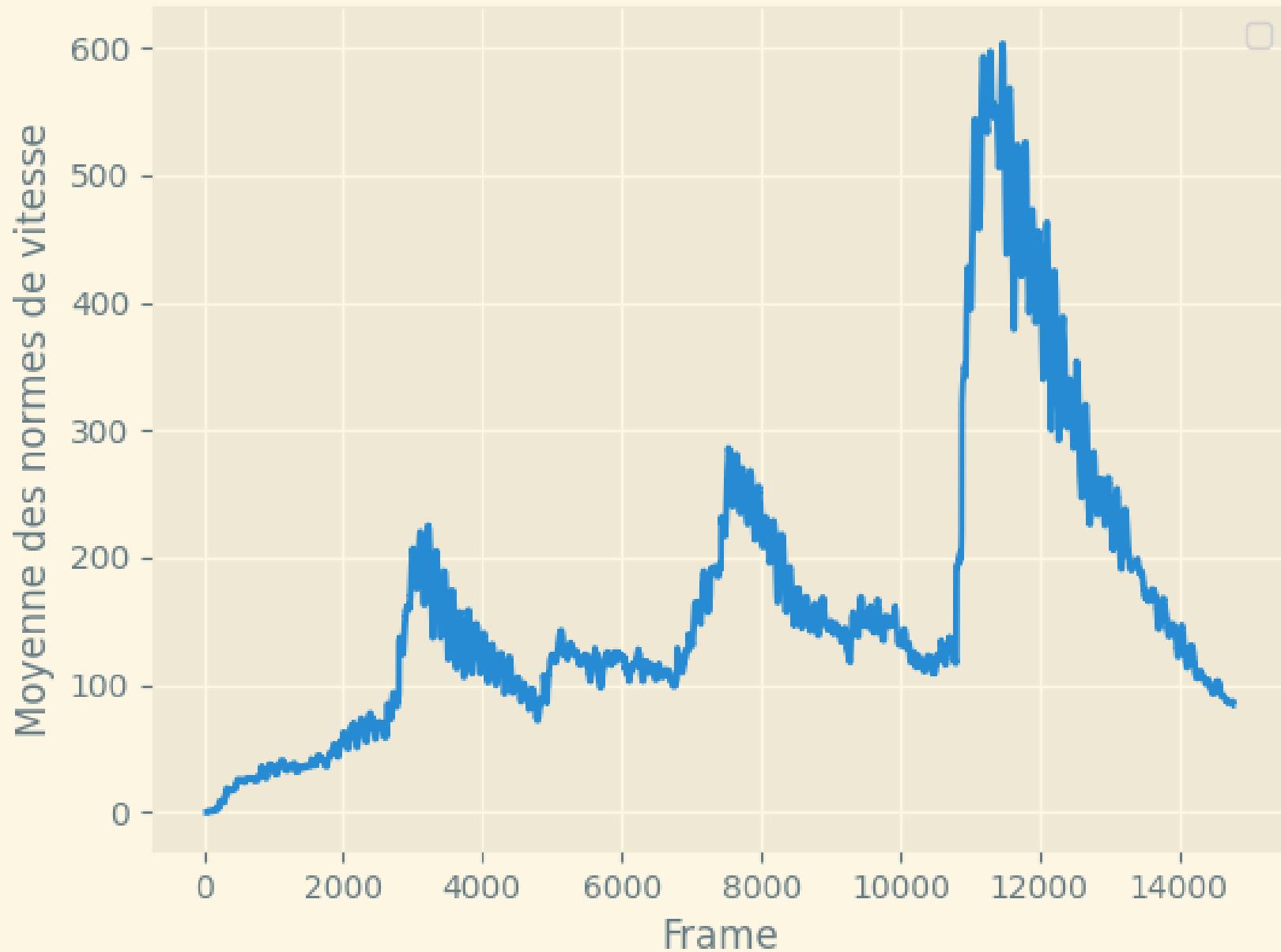
Force élastique en violet
Force fictive en noir

Déformation sous l'effet de la force fictive

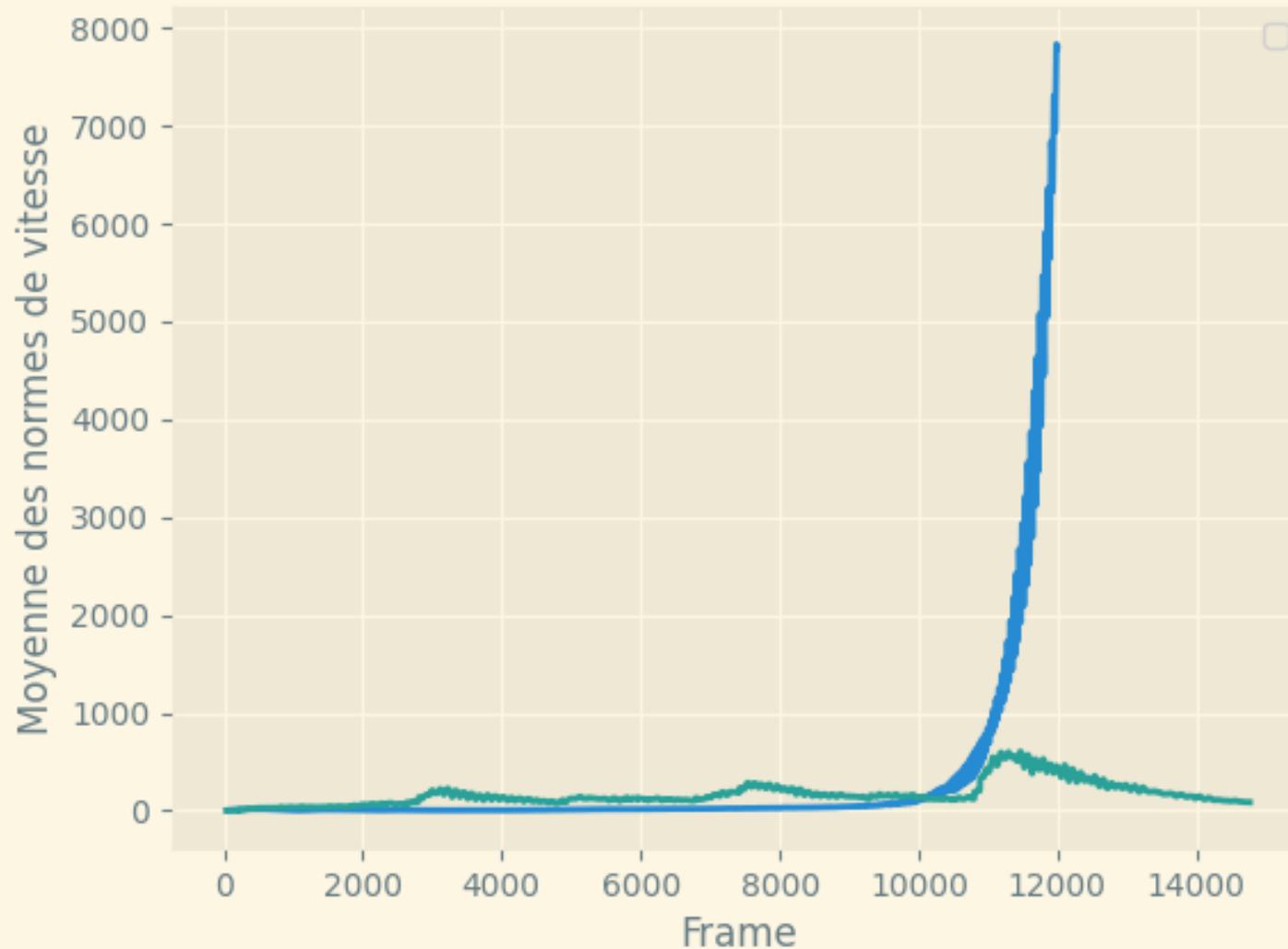


Désordre qui revient à une position d'équilibre

Résultat de la simulation avec RK4



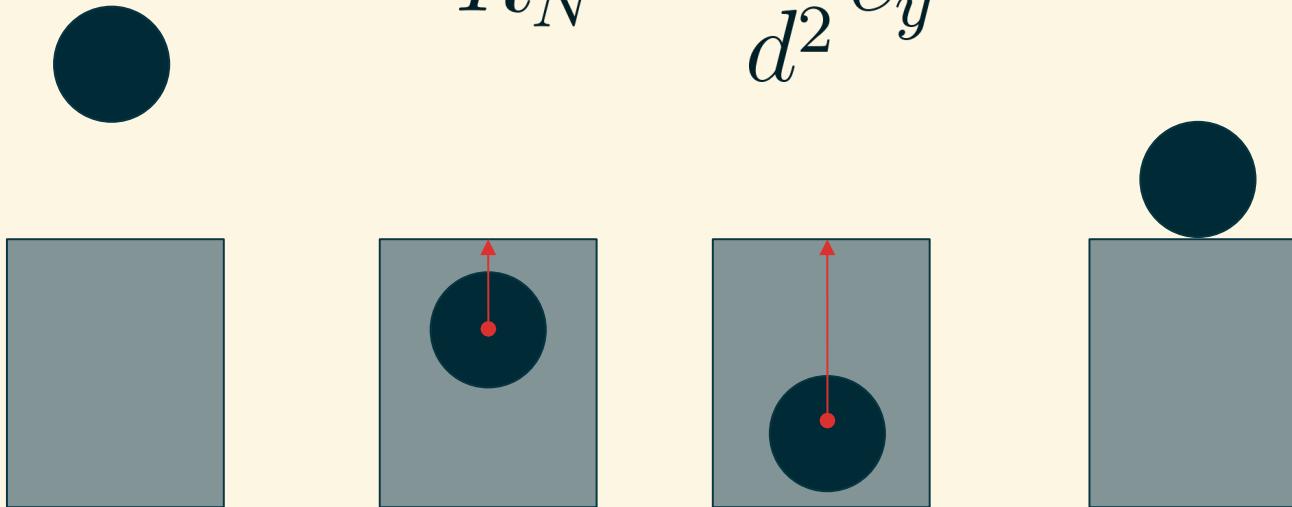
Comparaison des résultats des méthodes utilisées



2. Implémentation des collisions

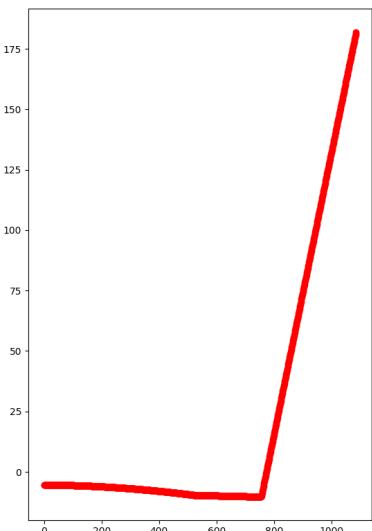
Essai avec collision “magnétique”

$$\vec{R}_N = \frac{K}{d^2} \vec{e}_y$$



Avantage majeur : nécessite peu de calcul et facile à implémenter

Résultats pour différentes valeurs de K pour la collision “magnétique”



$$K = 1.0$$

$$K = 1.0e - 1$$

$$K = 1.0e - 2$$

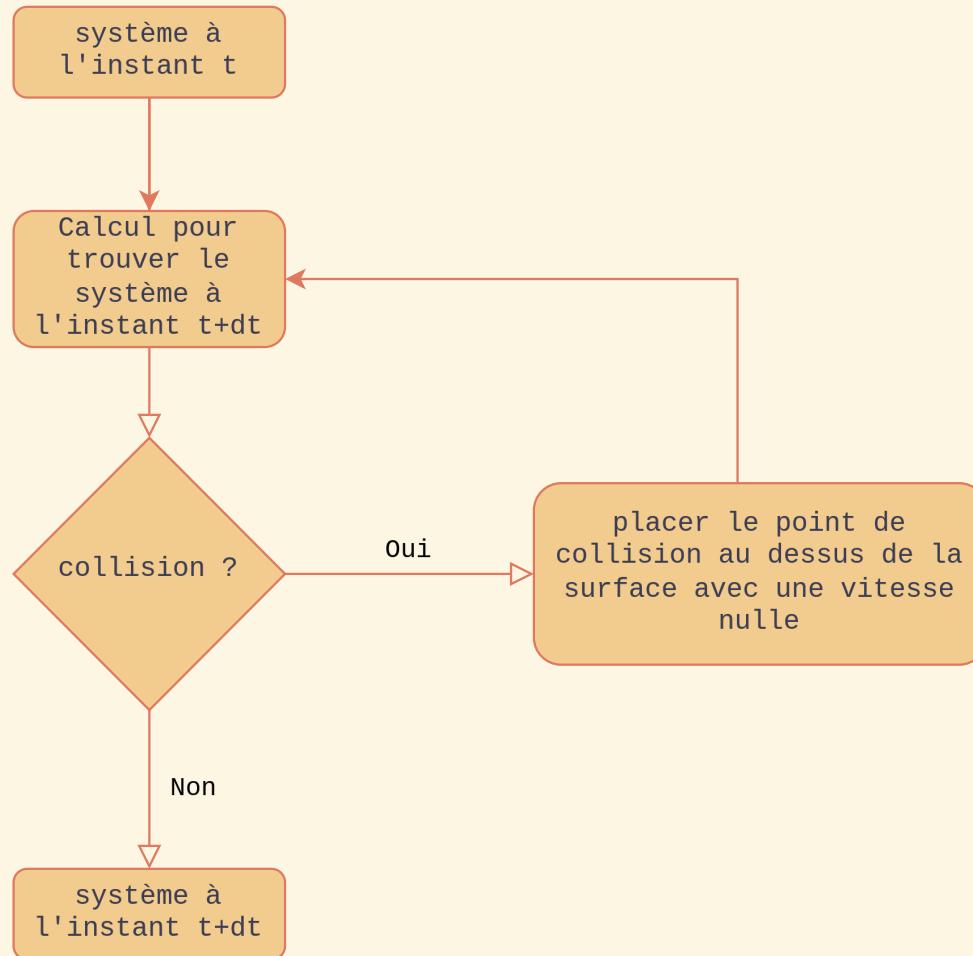
$$K = 1.0e - 4$$

Position moyenne verticale en fonction du temps

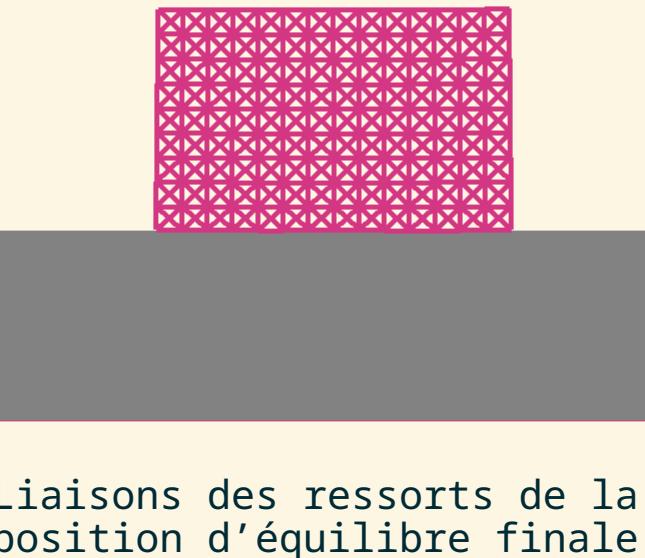
Résultats surprenants car :

- Force discontinue
- Force initialement très élevé lorsque $d = 0$

Collision via une méthode type Backtracking



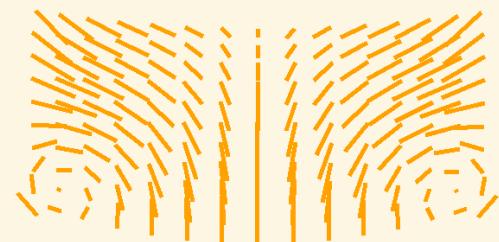
Résultats



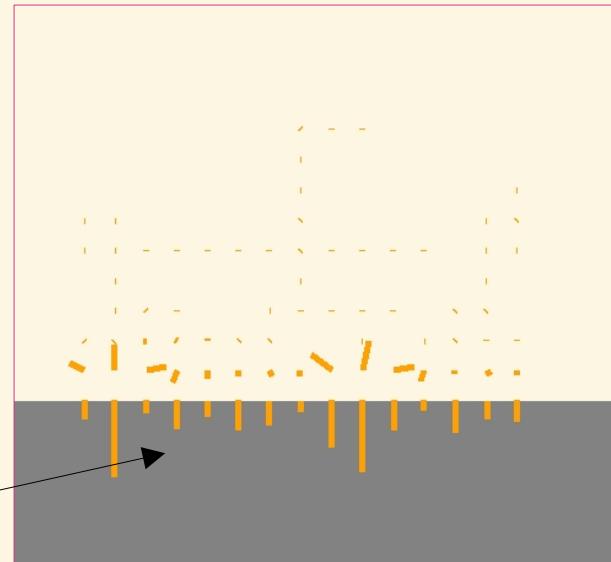
Liaisons des ressorts de la position d'équilibre finale

Oscillations dues
aux
replacements

Champ d'accélération lorsque le système rebondit en l'air

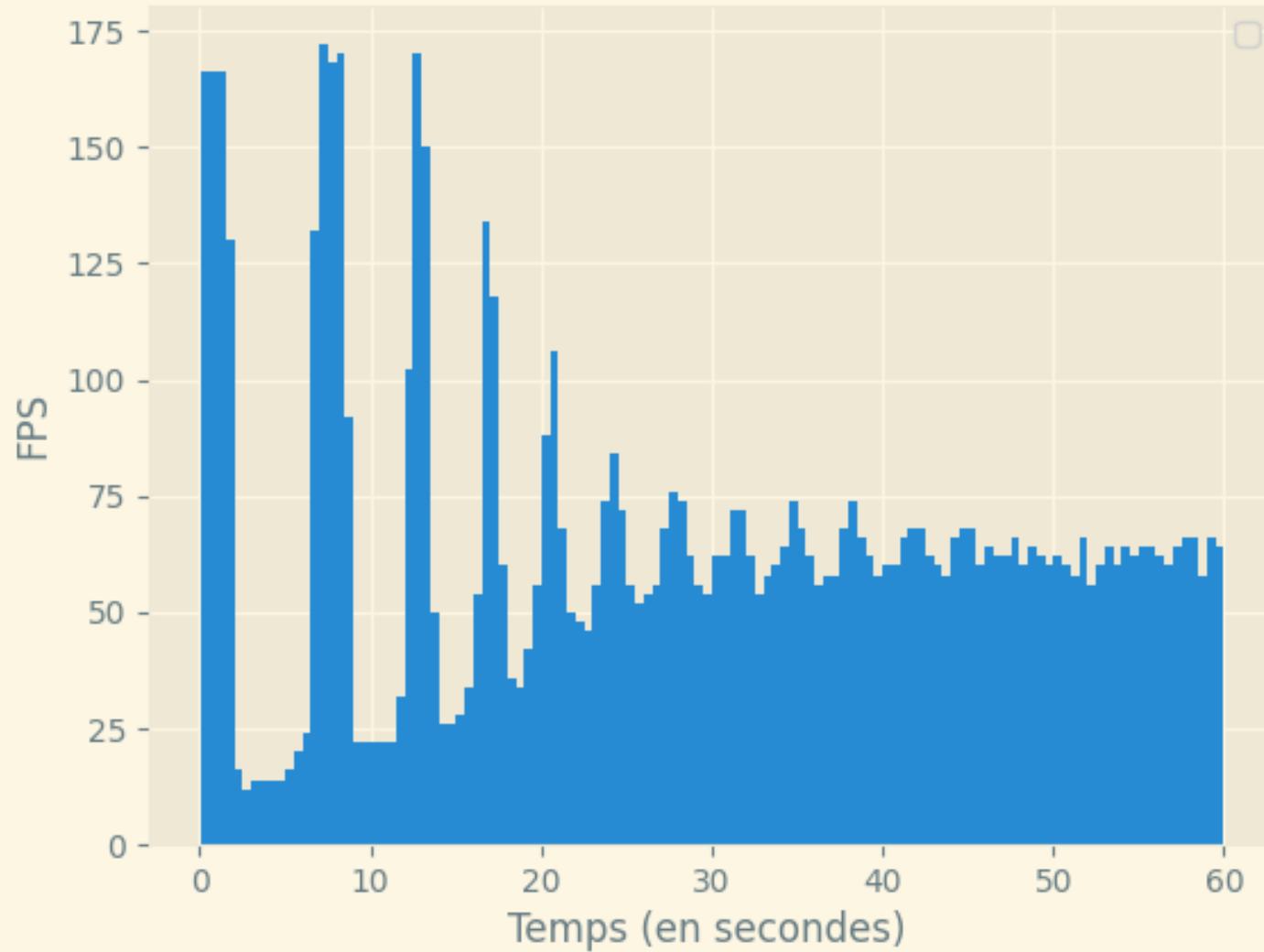


Champ d'accélération lorsque le système est au contact du sol



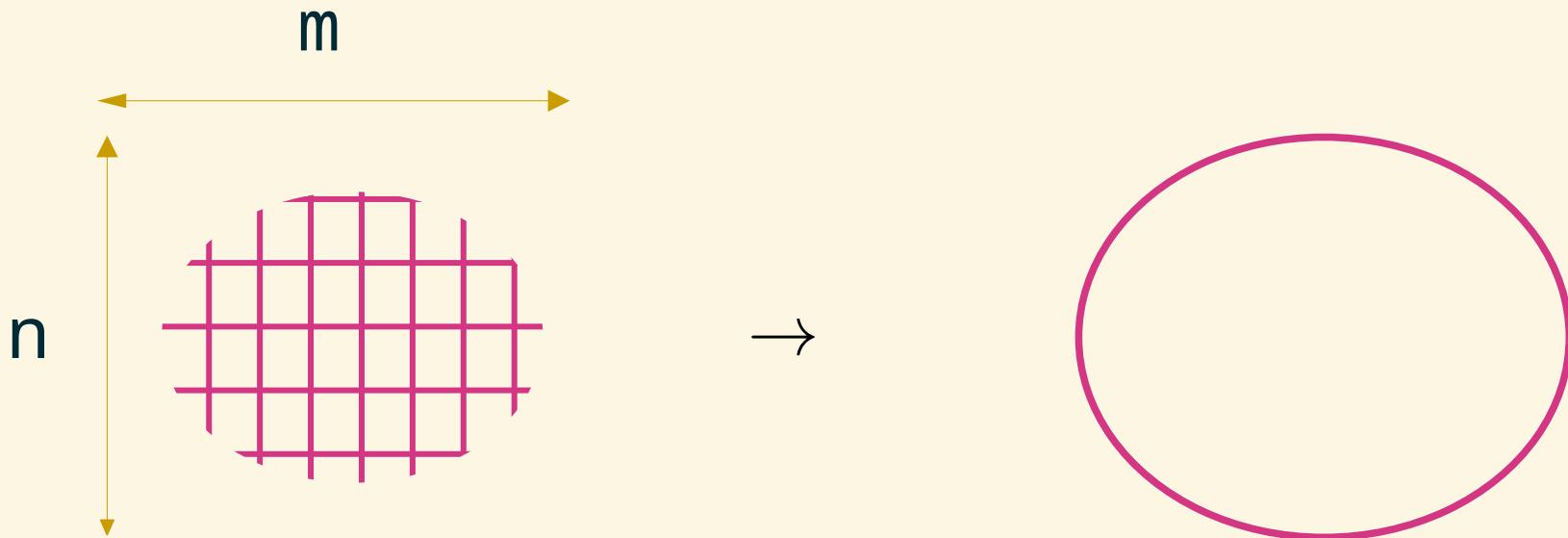
Résultats

Nombre de frames générées par secondes (FPS) durant la simulation



III. Deuxième approche

Modèle de la bulle de gaz parfait

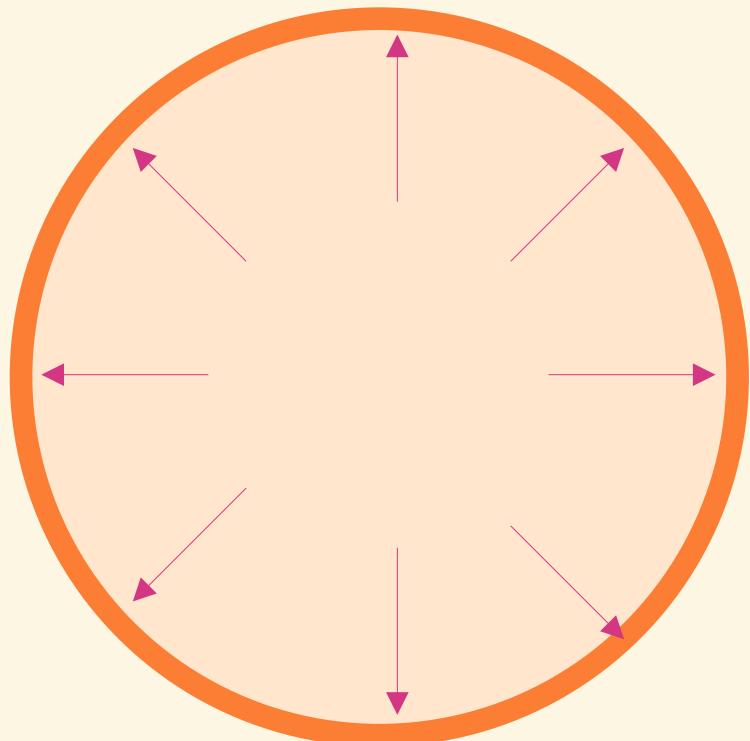


Complexité : $\mathcal{O}(m * n) \rightarrow \mathcal{O}(m + n)$

Avantages :

- Moins de particules au total à simuler
- Moins de particules au contact du sol

Modèle de la bulle de gaz parfait



$$\vec{F} = P d\vec{S}$$

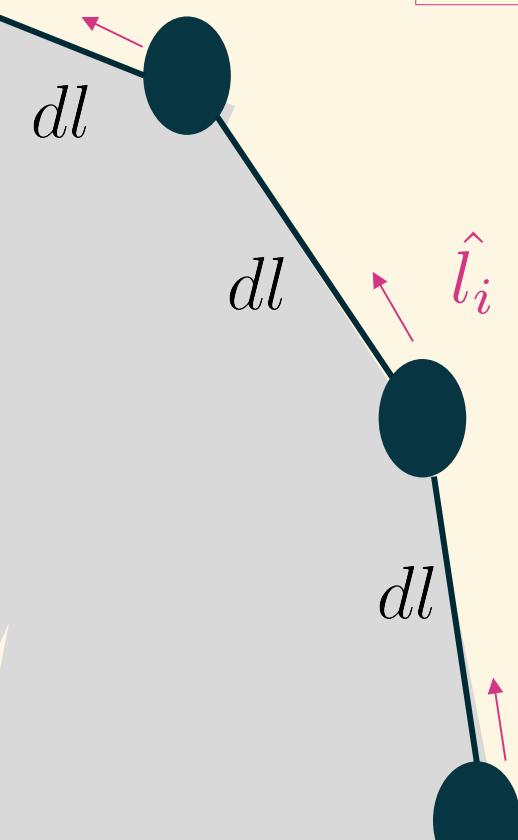
$$P = \frac{nRT}{V}$$

$$\boxed{\vec{F} = K_{nRT} \frac{1}{V} d\vec{S}}$$

Calcul du volume

Théorème de Gauss (conséquence d'Ostrogradski) :

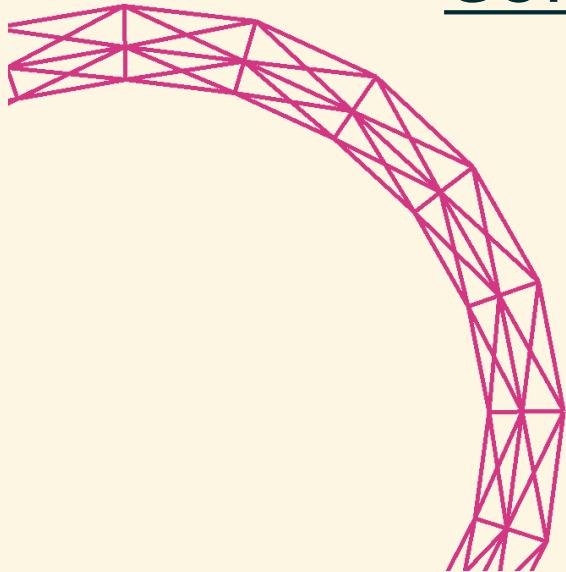
$$\iint_S \operatorname{div} \vec{F} \cdot d\vec{S} = \oint_C \vec{F} \cdot d\vec{l}$$



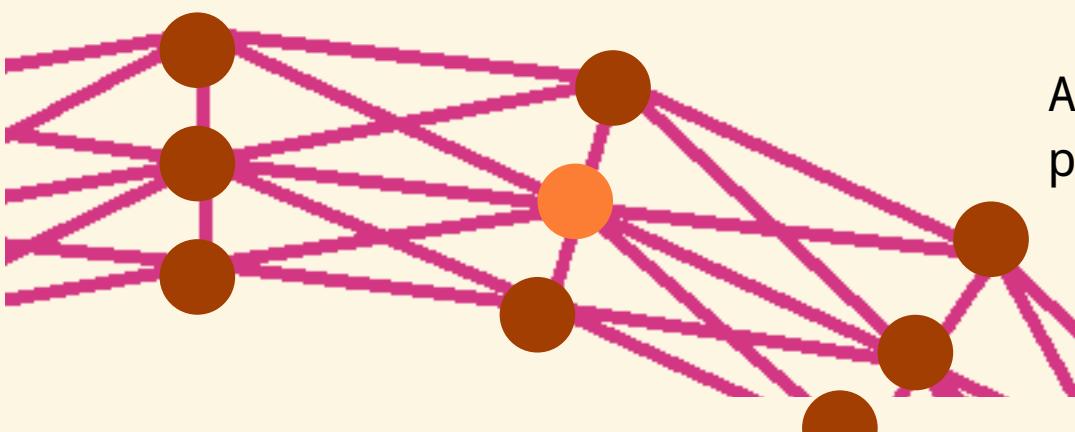
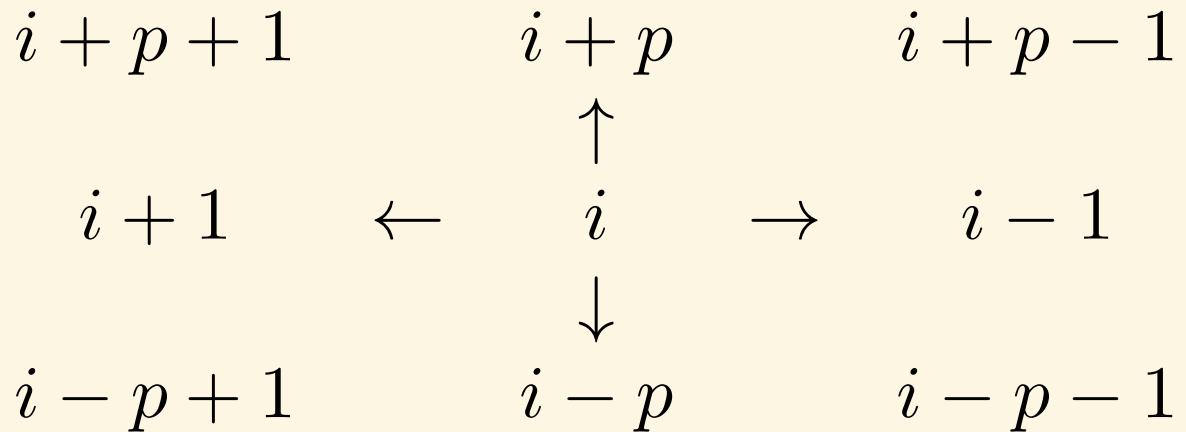
$$\iint_S \operatorname{div} \vec{F} \cdot d\vec{S} = S \quad \left| \begin{array}{l} \vec{F} = x \vec{e}_x \\ \operatorname{div} \vec{F} = 1 \\ \vec{F} \cdot d\vec{l} \\ = \vec{F} \cdot \hat{l} dl \\ = x \cdot \hat{l}_x \cdot dl \end{array} \right.$$

$$S \approx \sum x_i \cdot \hat{l}_{i,x} \cdot dl$$

Compromis ressort/gaz pour conserver la forme



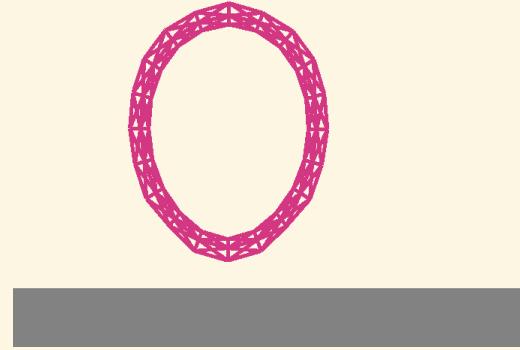
Les points sont stockés dans un tableau pour un accès en $O(1)$. La navigation se fait en suivant :



Avec p le nombre de points par couche

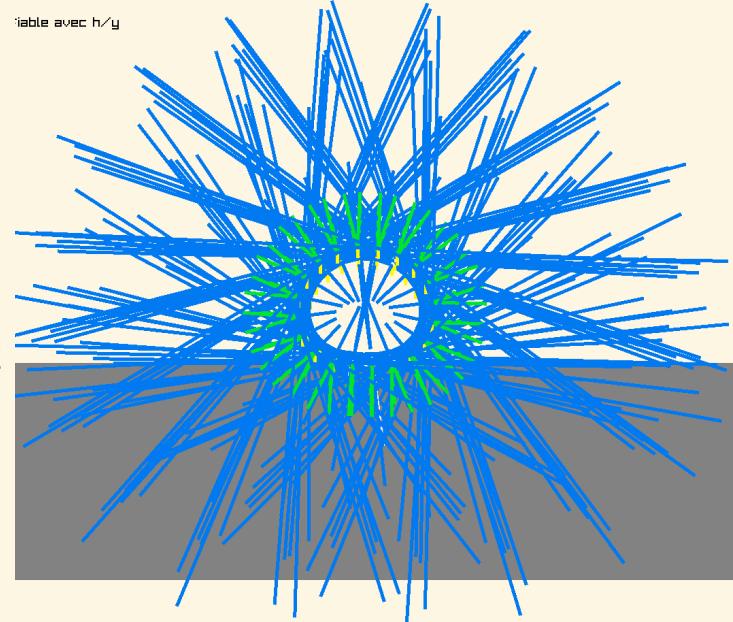
Résultat

Déformation après rebond

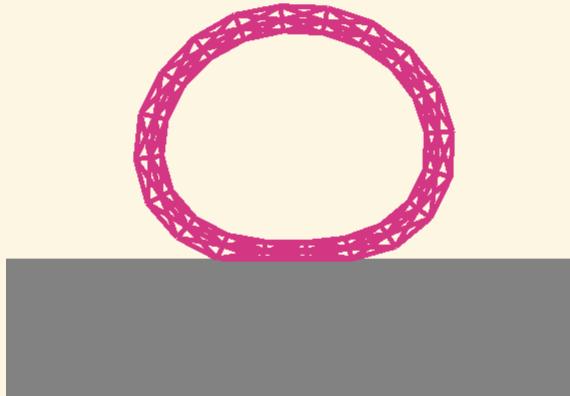


Force élastique en violet
Force du gaz en vert

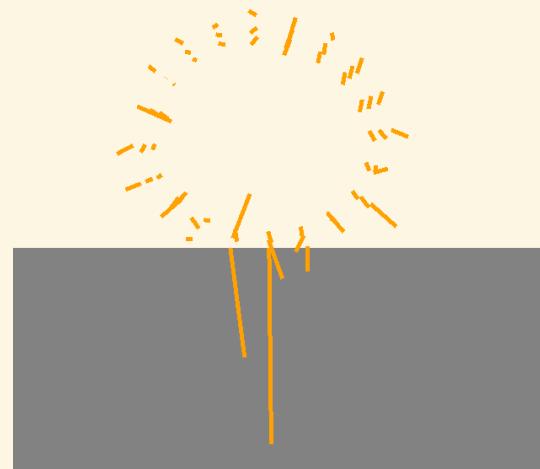
Forces appliquées sur les points



Position d'équilibre finale

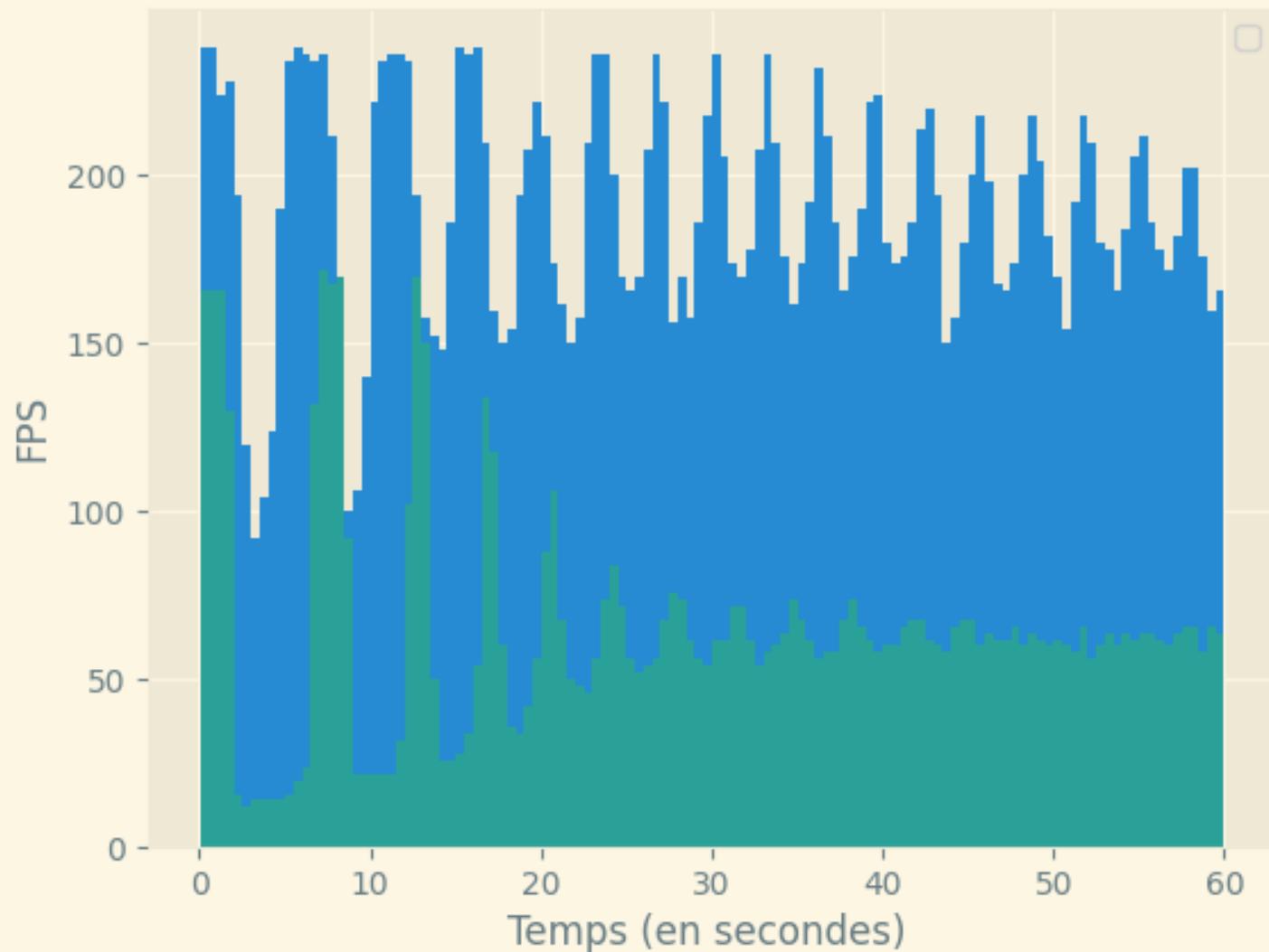


Profil d'accélération

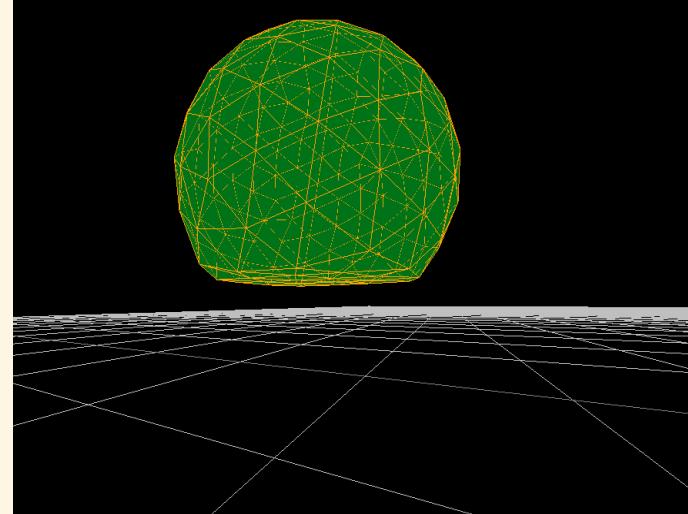
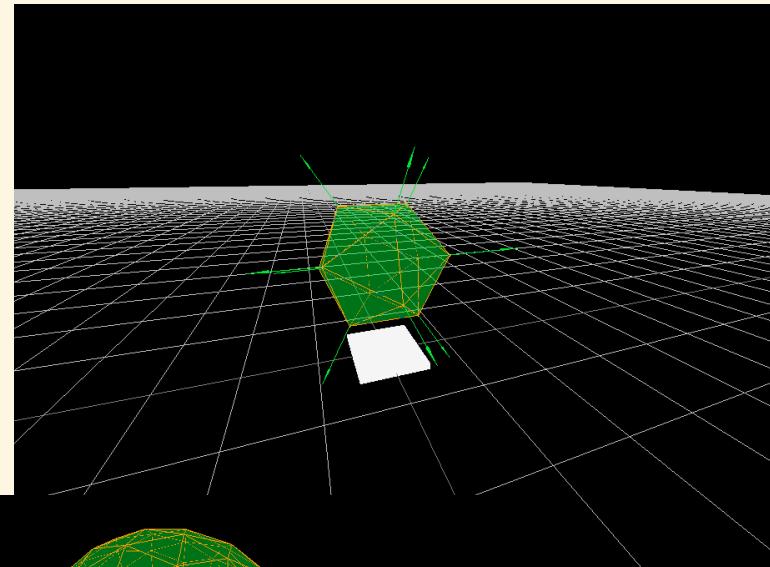
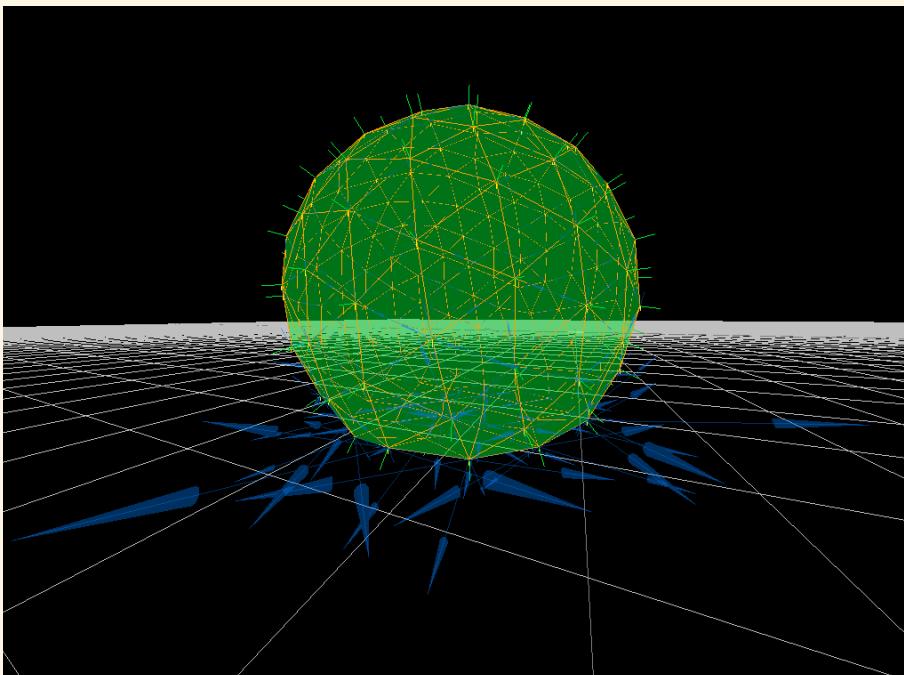


Résultats

Comparaison du nombre de frames générés par seconde via les deux méthodes (pour un disque ayant la hauteur du cube)



Extrapolation en 3 dimensions



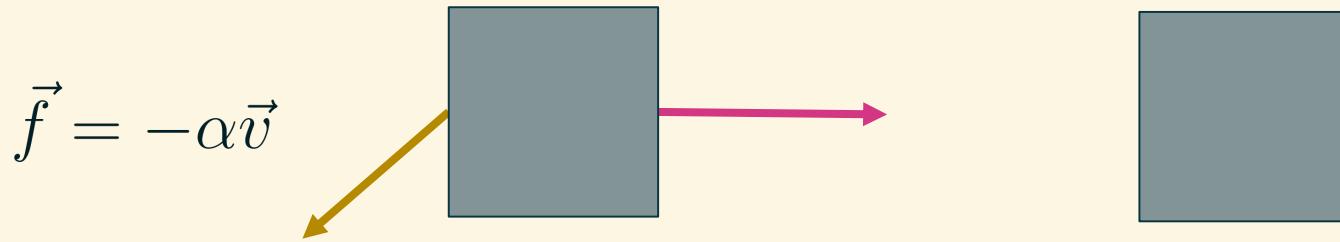
Conclusion

- Méthode masse-ressort pour simuler un comportement cohérent
- Méthode d'intégration de Runge et Kutta pour simuler un comportement stable
- Assimilation de la partie interne par un gaz parfait pour minimiser le temps de calcul

Annexe

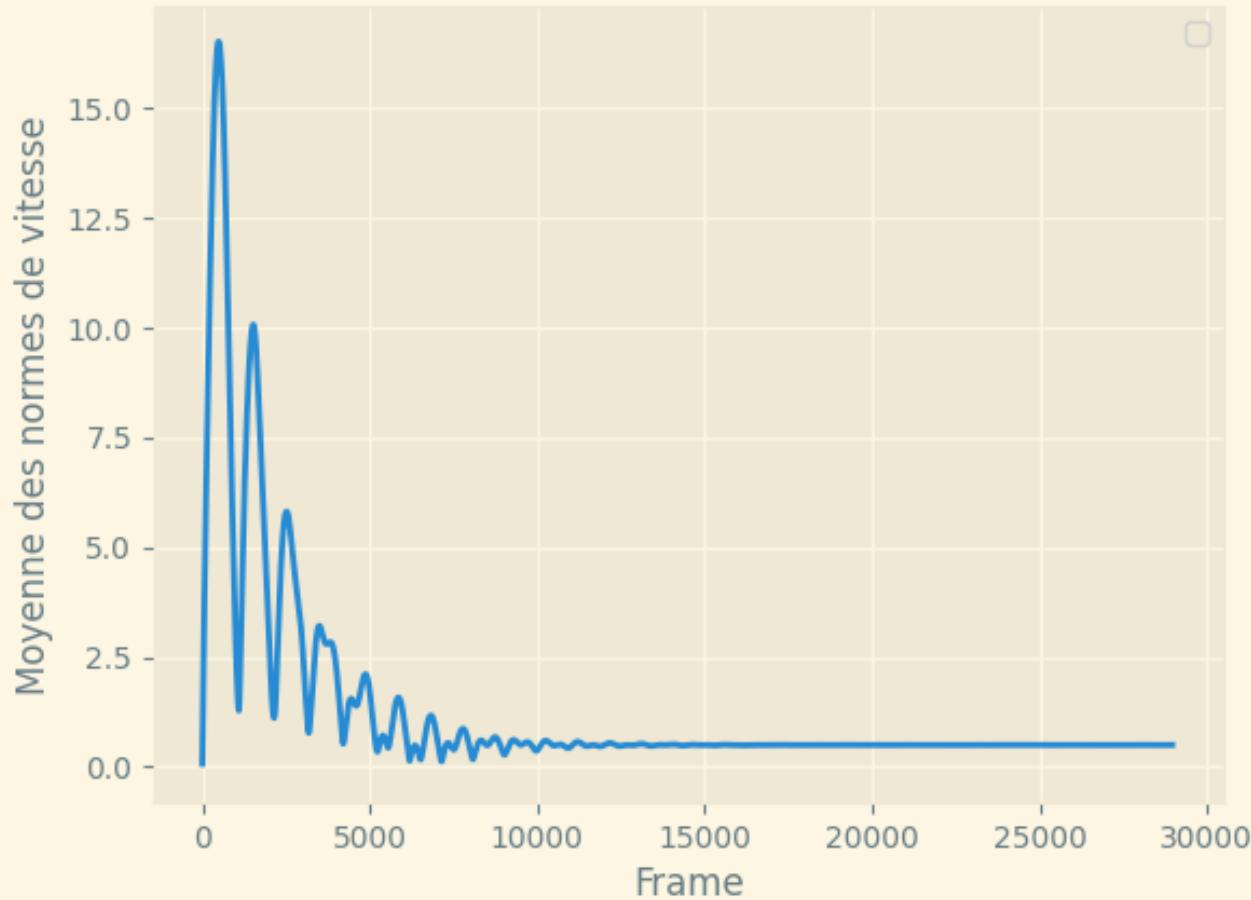
Méthode d'Euler implicite

$$\vec{R} = k * (l - l_0) \vec{e}_x$$

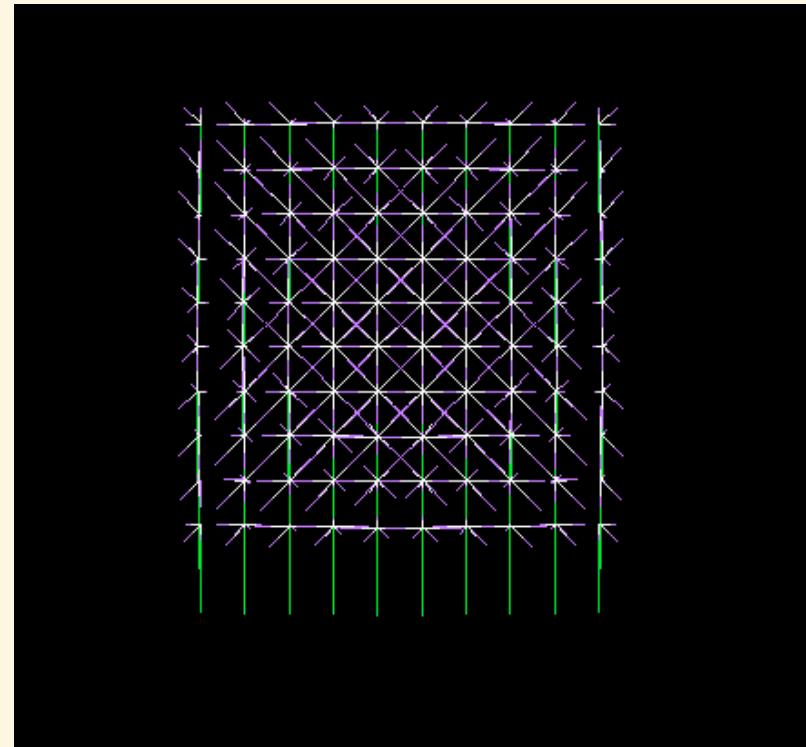
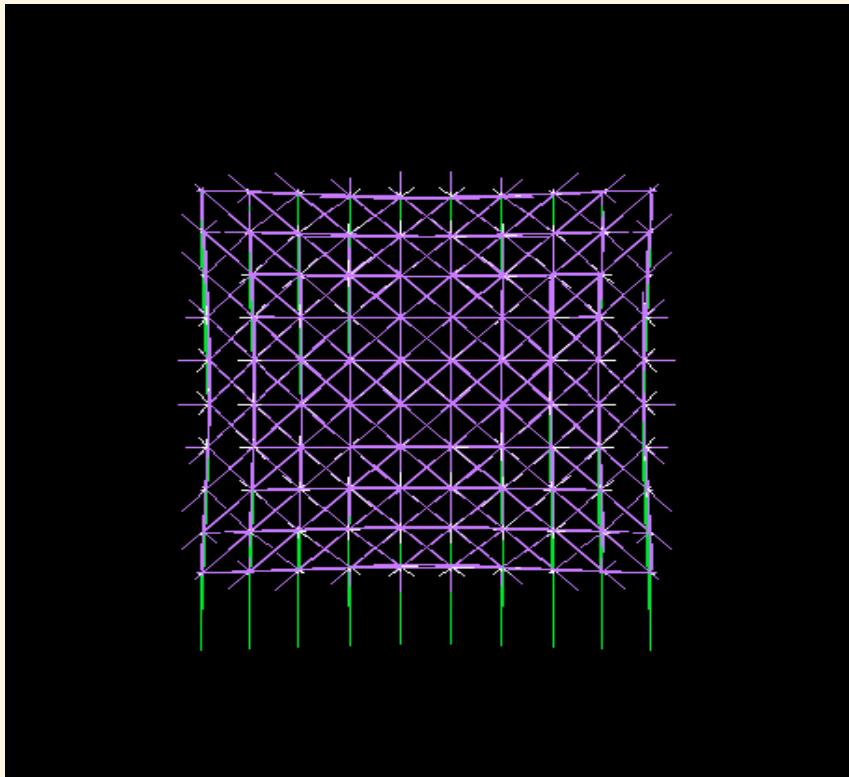


v[t + 1] = v[t] + acceleration(t) * dt
x[t + 1] = x[t] + v[t + 1] * dt

Méthode d'Euler implicite



Effet de respiration grâce au frottement



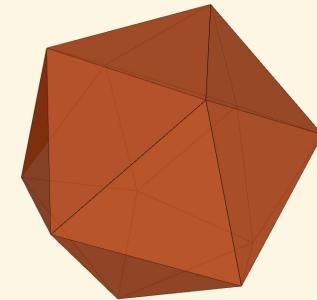
gravité

frottement
amortisseur

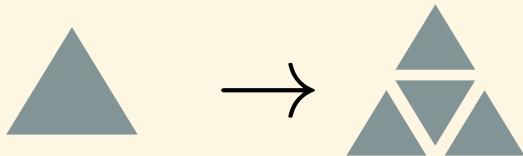
force de ressort
élastique

Construction de l'icosphère

On démarre avec un icosaèdre (20 faces) :



Pour chacune des faces, on applique la transformation :



Puis on norme les vecteurs des sommets pour qu'ils soient distants de R avec le centre

On itère ce procédé pour diminuer la rugosité de la sphère

Il y aura ainsi 20×3^n sommets après n itérations

Description d'un solide isotrope : Loi de Hooke et équation de Lamé

→ La loi de Hooke

$$\sigma = E\varepsilon$$

où

σ La contrainte (pression)

E Le module de Young

ε L'allongement relatif à la longueur à vide

$$(\varepsilon = \frac{l - l_0}{l_0})$$

→ L'équation de Lamé

$$\sigma = E\varepsilon \iff \sigma = \lambda \text{Tr}(\varepsilon) + 2\mu\varepsilon$$

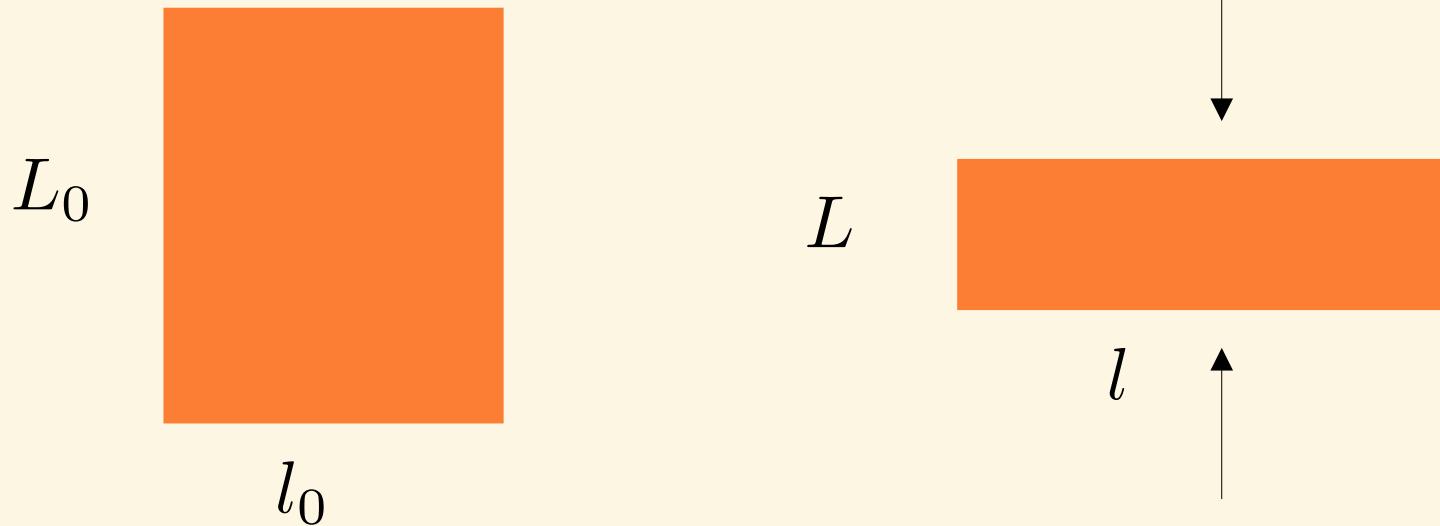
En posant : $E = \frac{(3\lambda+2\mu)\mu}{\lambda+\mu}$

Le couple (λ, μ)
défini entièrement
le matériel isotrope

Deuxième paramètre pour décrire un matériel isotrope : coefficient de Poisson

$$\nu = \frac{\lambda}{2(\lambda + \mu)}$$

$$\nu = \frac{\frac{l_0 - l}{l_0}}{\frac{L_0 - L}{L_0}}$$



Avec le module de Young (E) et le coefficient de Poisson (ν), on retrouve facilement les coefficients de Lamé.

Preuve de la stabilité de Runge et Kutta

Une méthode d'intégration est dite stable de constante S si il existe S tel que, si

$$\begin{aligned}y_{n+1} &= y_n + h\phi(t_n, y_n) \\ \tilde{y}_{n+1} &= \tilde{y}_n + h\phi(t_n, \tilde{y}_n) + \varepsilon_n\end{aligned}$$

alors

$$\max_{n \leq N} |y_n - \tilde{y}_n| \leq S \sum_{n \leq N} |\varepsilon_n|$$

Où N est le nombre d'étapes durant l'entièreté de la simulation

Preuve de la stabilité de Runge et Kutta

Lemme : Si une solution au problème de Cauchy y est k-lipschitzienne, on a avec le lemme de Gronwall :

ϕ est Λ -lipschitzienne \implies
la méthode d'intégration est stable,
de constante de stabilité $S = e^{\Lambda T}$

Où T est le temps total de la simulation

Preuve de la stabilité de Runge et Kutta

$$t_{n,i} = t_n + c_i \times h$$

$$z(t_{n,i}) = z(t_n) + h \sum_{j < i} a_{i,j} f(t_{n,j}, z(t_{n,j}))$$

$$z(t_{n+1}) = z(t_n) + h \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j}))$$

$$\phi(t, z) = \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j}))$$

Reste à prouver que ϕ est lipschitzienne. On part de deux sources y et z quelconques dans \mathbb{R} :

Lemme : $\forall i,$
 $|y_i - z_i| \leq \sum_{p=0}^i (\alpha k h)^p |y - z|$

Avec

$$\alpha = \max \sum |a_{i,j}|$$

Preuve de la stabilité de Runge et Kutta

$$\begin{aligned}t_{n,i} &= t_n + c_i \times h \\z(t_{n,i}) &= z(t_n) + h \sum_{j < i} a_{i,j} f(t_{n,j}, z(t_{n,j})) \\z(t_{n+1}) &= z(t_n) + h \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j})) \\\phi(t, z) &= \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j}))\end{aligned}$$

d'où

$$\begin{aligned}|\phi(t, y) - \phi(t, z)| &\leq \sum |b_j| k |y_j - z_j| \\&\leq k \sum |b_j| \sum_{p=0}^j (\alpha kh)^p |y - z| \\&\leq \Lambda |y - z|\end{aligned}$$

Preuve de la stabilité de Runge et Kutta

$$t_{n,i} = t_n + c_i \times h$$

$$z(t_{n,i}) = z(t_n) + h \sum_{j < i} a_{i,j} f(t_{n,j}, z(t_{n,j}))$$

$$z(t_{n+1}) = z(t_n) + h \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j}))$$

$$\phi(t, z) = \sum_{j \leq 4} b_j f(t_{n,j}, z(t_{n,j}))$$

Preuve du lemme par récurrence

$$\begin{aligned} & |y_i - z_i| \\ & \leq |y - z| + h \sum_{j \leq i} |a_{i,j}| |f(t_j, y_j) - f(t_j, z_j)| \\ & \leq |y - z| + h \sum_{j \leq i} |a_{i,j}| k |y_j - z_j| \\ & \leq |y - z| + \alpha h k \max |y_j - z_j| \\ & \leq |y - z| + \alpha h k \sum (\alpha h k)^p |y_j - z_j| \\ & \leq \sum_{p=0}^i (\alpha h k)^p |y - z| \end{aligned}$$

```
open Raylib

type 'a graph = 'a Array.t
type listadj = int list array

type vec = float * float * float

type point = {
  pos : vec;
  vit : vec;
  mass : float;
}

type force = vec * Color.t

(*arguments de la fonction somme des forces*)
type arguments = {
  points : point graph;
  k_ressort : float;
  penche : bool;
  center : vec;
}

(*état actuel de la simulation*)
type status = {
  blob : point graph; (*le tableau des points*)
  t : int; (*temps*)
  k_ressort : float; (*la constante de ressort*)
  penche : bool; (*5.0 ou 0.0 selon si la gravité est inclinée ou non*)
  cam : Camera3D.t; (*la camera*)
}
```

```

maths.ml

open Types
let foI = float_of_int
let iof = int_of_float

let fst {a,b,_} = a
let snd {_,b,c} = b
let trd {_,_,c} = c

let r3_to_vec3 (x,y,z) = Raylib.Vector3.create x z y
let (+$) (a,b,c) (x,y,z) = {a+.x, b+.y, c+.z}
let (-$) (a,b,c) (x,y,z) = {a-.x, b-.y, c-.z}
let (*$) (a,b,c) q = {a*.q, b*.q, c*.q}
let (/$_) (a,b,c) q = (a/.q, b/.q, c/.q)

let zero = 0.0,0,0,0,0
let zero_r = Raylib.Vector3.create 0.0 0.0 0.0

(*dot produit fin le produit scalaire quoi*)
let ps (a,b,c) (x,y,z) = a*x +. b*y +. c*z

(*cross produit produit vectoriel*)
let pv (a,b,c) (x,y,z) = (b*.z -. c*.y, c*.x -. a*.z, a*.y -. b*.x)

let abs_f x = if x < 0.0 then -x else x
let sign_f x = if x < 0.0 then -1.0 else 1.0

let dist_square (xa,ya,za) (xb,yb,zb) = (xa-.xb)**2.0 +. (ya-yb)**2.0 +. (za-.zb)**2.0
let dist a b = sqrt (dist_square a b)
let norme = dist zero

let vect_elem a b = (b-$ a) /$ dist a b
let shmidtz = vect_elem zero

let det3 (a,b,c) (d,e,f) (g,h,i) =
  a *. (e*.i -. h*.f) -. d *. (b*.i -. h*.c) +. g *. (b*.f -. e*.c)

let order a b c center =
  (*on veut que a b c soit dans le sens horraire vus depuis le centre center*)
  let det = det3 (a-$center) (b-$center) (c-$center) in
  if det > 0.0 then a,b,c
  else a,c,b

(* vecteur normal à la surface a b c vu depuis le centre center*)
let normal a b c center =
  let a,b,c = order a b c center in
  pv (b-$a) (c-$a)

(* fonctions modifiables pour plus de détail*)
let collision (_,_,z) =
  z < 0.01

let fix_collision (x,y,_) =
  (x,y,0.01)

let somme_forces l p =
  let s = List.fold_left (fun x (f,_) -> x +$ f) zero l in
  if collision p.pos then zero else s

(*l'aire d'un triangle situé sur les sommets a b c*)
let area (a,b,c) = norme (pv (b-$a) (c-$a)) /. 2.0

(*On approxime le volume de l par le volume d'une ellipsoide*)
let volume l =
  let l = Array.map (fun x-> x.pos) l in
  let minx,miny,minz = Array.fold_left
    (fun (a,b,c) (x,y,z) -> (min a x, min b y, min c z)) (max_float,max_float,max_float) l
  in
  let maxx,maxy,maxz = Array.fold_left
    (fun (a,b,c) (x,y,z) -> (max a x, max b y, max c z)) (min_float,min_float,min_float) l
  in
  4.0/3.0 *. 3.14 *. (maxx-.minx) *. (maxy-.miny) *. (maxz-.minz)

```

```

icosphere.ml

open Maths
open Constantes

(*renvoie la liste contenant les mêmes éléments mais de façon unique*)
let unique l =
  let h = Hashtbl.create 10 in
  List.iter (fun x -> Hashtbl.replace h x ()) l;
  Hashtbl.to_seq_keys h |> List.of_seq

let subdivide positions triangles =
  (* rajoute du détail à une sphère de rayon 1 *)
  let q = Array.length positions in (* nombre de points actuels *)
  let edges = (* propriété 1 : les arrêtes a --- b sont telles que a < b *)
    List.map (fun (i,j,k) -> [(i,j);(i,k);(j,k)]) triangles |> List.concat |> unique
  in
  (* tous les points entre les sommets i et j*)
  let middle = Hashtbl.create 100 in
  let newverticieslist = List.mapi
    (fun i (a,b) ->
      Hashtbl.replace middle (a,b) (q+i);
      shmidtz (positions.(a) +$ positions.(b)))
  ) edges in
  let positions' = Array.concat [positions;Array.of_list newverticieslist] in
  let triangles' =
    List.map
      (fun (a,b,c) ->
        let d = Hashtbl.find middle (a,b) in
        let e = Hashtbl.find middle (a,c) in
        let f = Hashtbl.find middle (b,c) in
        (* on renvoie les 4 triangles formés par le triangle abc,
           tout en conservant la propriété 2 grâce à la propriété 1*)
        [(b,d,f);(a,d,e);(c,e,f);(d,e,f)])
    )
  triangles |> List.concat
in positions', triangles'

(*icosaèdre brut*)
let icosahedron =
  let x = 0.525731112119133606 in
  let z = 0.850650808352039932 in
  []
  [|
    {-.x, 0.0, z}; {x, 0.0, z}; {-x, 0.0, -.z};
    {x, 0.0, -.z}; {0.0, z, x}; {0.0, z, -.x};
    {0.0, -.z, x}; {0.0, -.z, -.x}; {z, x, 0.0};
    {-z, x, 0.0}; {z, -.x, 0.0}; {-z, -.x, 0.0}
  |]

(*Propriété 2 : un triangle est une liste de 3 indices de sommets, dans l'ordre croissant*)

let indices_triangles = [
  {0, 1, 4}; {0, 4, 9}; {4, 5, 9};
  {4, 5, 8}; {1, 4, 8}; {1, 8, 10};
  {3, 8, 10}; {3, 5, 8}; {2, 3, 5};
  {2, 3, 7}; {3, 7, 10}; {6, 7, 10};
  {6, 7, 11}; {0, 6, 11}; {0, 1, 6};
  {1, 6, 10}; {0, 9, 11}; {2, 9, 11};
  {2, 5, 9}; {2, 7, 11}
]

let icosphere, indices_triangles =
  let rec aux n (positions,triangles) =
    if n = 0 then positions,triangles
    else aux (n-1) (subdivide positions triangles)
  in aux deep (icosahedron,indices_triangles)

let n = Array.length icosphere

```

```

graph.ml

open Constantes
open Maths
open Types

let fold_left = Array.fold_left
let mapi = Array.mapi
let map = Array.map
let map2 = Array.map2
let init = Array.init
let iteri = Array.iteri

let ( *%) q = map (fun t -> t *$ q)
let ( +%) = map2 (+$)
let ( -%) = map2 (-$)

(*la figure intiale *)
let initial () =
  Array.map
    (fun (x,y,z) ->
      let r = rayon in
      {
        pos = x*.r,y*.r,z*.r +. rayon +. hauteur_initiale;
        vit = 0.0,0.0,0.0;
        mass = mass;
      })
  Icosphere.icosphere

let triangles_with i l center =
  (*renvoie la liste des aires et des vecteurs normaux des triangles voisins à i*)
  let is_in (a,b,c) = a = i || b = i || c = i in
  List.filter is_in Icosphere.indices_triangles
  |> List.map (fun (i,j,k) -> area (l.(i).pos,l.(j).pos,l.(k).pos), normal l.(i).pos l.(j).pos l.(k).pos center)

(*tableau de liste d'adjacence, avec la distance d'équilibre*)
let g =
  let l = initial () in
  let g = Array.make Icosphere.n [] in
  List.iter
    (fun (i,j,k) ->
      g.(i) <- j::g.(i); g.(i) <- k::g.(i);
      g.(j) <- i::g.(j); g.(j) <- k::g.(j);
      g.(k) <- i::g.(k); g.(k) <- j::g.(k))
  Icosphere.indices_triangles;
  let g = Array.map Icosphere.unique g in
  Array.mapi
    (fun i l' ->
      List.map (fun j -> j, dist l.(i).pos l.(j).pos)
      l')
  g

(*renvoie les sommets adjacents à i et la longeur à vide*)
let linked_to l i = List.map (fun (j,10) -> l.(j), 10) g.(i)

(*creation d'un graph a partir d'un tableau de positions et de vitesses*)
let to_points y y' blob =
  init Icosphere.n (fun i-> {pos=y.(i); vit = y'.(i); mass = blob.(i).mass})

(*renvoie la liste des triangles*)
let surfaces l = List.map (fun (i,j,k) -> l.(i),l.(j),l.(k)) Icosphere.indices_triangles

```

```
open Maths
open Raylib.Color
open Constantes
open Types

(*force elastique avec les voisins *)
let ressort src dst 10 k_ressort = vect_elem dst.pos src.pos *$  
  (-.k_ressort *. (dist src.pos dst.pos -.10))

(*force de repulsion avec les voisins*)
let repulsion src dst = vect_elem dst.pos src.pos *$  
  (k_repulsion/. (dist src.pos dst.pos)**4.0)

(*force d'amortissement avec les voisins *)
let amortisseur src dst =  
  let er = vect_elem src.pos dst.pos in  
  er *$ (k_damping *. ps (src.vit -$ dst.vit) er)

(*force de gonflement du gaz*)
let gaz id vol blob center=  
  List.fold_left (+$) zero  
  (List.map (fun (surface,norm) -> shmidtz norm *$ (nRT *. surface /. vol))  
  (Graph.triangles_with id blob center))

(*force induite par le champs gravitationnel*)
let poids penche src =  
  (gravity +$ if penche then (5.0,0.0,0.0) else zero) *$ (1.0*.src.mass)

(*renvoie une liste de couples (force, couleur)*)
let bilan_des_forces src id volume blob {penche;k_ressort;center;_} =  
  [  
    poids penche src, yellow;  
    gaz id volume blob center, green  
  ]  
  @ List.concat  
  (  
    List.map  
    (fun (dst,10) ->  
      [  
        ressort src dst 10 k_ressort, Raylib.fade blue 0.4;  
        amortisseur src dst, raywhite;  
        repulsion src dst, red;  
      ]  
    )  
    (Graph.linked_to blob id)  
  )
```

```

open Maths
open Constantes
open Graph
open Types

(*fonction qui donne l'acceleration en fonction de dt, la position et la vitesse*)
let f _ y y' args =
let blob = to_points y y' args.points in
let vol = volume args.points in
Graph.mapi
  (fun i p -
   if collision p.pos then
   begin
     y.(i) <- fix_collision p.pos;
     y'.(i) <- zero
   end;
   (somme_forces (Force.bilan_des_forces p i vol blob args) p ) /$ p.mass
  )
blob

let runge_kunta args =
let h = dt in
let h2 = h/.2.0 in
let hh4 = h*.h/.4.0 in
let h6 = h/.6.0 in
let y = Graph.map (fun x -> x.pos) args.points in
let y' = Graph.map (fun x -> x.vit) args.points in
let k1 = f 0.0 y y' args in
let k2 = f h2 (y +% h2 *% y') (y' +% h2 *% k1) args in
let k3 = f h2 (y +% h2 *% y' +% hh4 *% k1) (y' +% h2 *% k2) args in
let k4 = f h (y +% h *% y' +% (h*.h2) *% k2) (y' +% h *% k3) args in
let ny = y +% h *% y' +% (h*.h6) *% (k1 +% k2 +% k3) in
let ny' = y' +% h6 *% (k1 +% 2.0 *% k2 +% 2.0 *% k3 +% k4) in
to_points ny ny' args.points

(*diminuer dt avant d'utiliser ces méthodes*)
let verlet args =
let y' = Graph.map (fun x -> x.vit) args.points in
let prec = Graph.map (fun x -> x.pos) args.points in
let current = prec +% (dt *% y') in
let next = 2.0 *% current -% prec +% (dt*.dt *% f dt current y' args) in
to_points next ((1.0 /.dt) *% (next -% current)) args.points

let euler_explique args =
let y = Graph.map (fun x -> x.pos) args.points in
let y' = Graph.map (fun x -> x.vit) args.points in
let next = y +% dt *% y' in
let next' = y' +% dt *% f dt y y' args in
to_points next next' args.points

let integrate args =
if not animate then args.points else
match meth with
| "rk" -> runge_kunta args
| "euler" -> euler_explique args
| "verlet" -> verlet args
| _ -> failwith "methode inconnue"

```

```
(* --- Affichage --- *)
let fac_newt = 1e-1 (*augmente la norme des vecteurs lors de l'affichage*)
let w = 16 * 120
let h = 9 * 120

(* --- Constantes d'intégration --- *)
let meth = "rk" (*euler,rk,verlet*) (*méthode d'intégration*)
let animate = true
let dt = 1.0/.60.0 (*pas de temps (s)*)

(* --- Constantes physiques --- *)
let mass = 0.25 (*masses des particules (kg)*)
let k_ressort = 750.0 (*constante de raideur (N/m)*)
let k_damping = 1e-4 (*constante d'amortissement du ressort*)
let k_repulsion = 10.0 (*constante en k_rep/r**4*)
let gravity = (0.0,0.0,-9.81) (*champ gravitationnel m.s-2*)
let nRT = 1e4 (*facteur de corrélation entre la pression et le volume*)
let hauteur_initiale = 100.0 (*hauteur initiale du centre de la sphère*)

(* --- Taille du blob --- *)
let rayon = 10.0 (*rayon de la sphère*)
let deep = 1 (*nombre d'iteration en detail pour la sphère ico*)
```

```

main.ml
open Raylib
open Maths
open Graph
open Force
open Types
open Constantes
open Icosphere

(*affiche les points ou les forces*)
let draw st_center =
  let draw_tri a b c =
    let a,b,c = order a b c center in
    draw_triangle_3d (r3_to_vec3 a) (r3_to_vec3 b) (r3_to_vec3 c)
  in
  let draw_vec src f col =
    let f = f *$ fac_newt in
    let end_force = src +$ f in
    let mid_force = src +$ (f *$ 0.75) in
    begin
      draw_line_3d (r3_to_vec3 src) (r3_to_vec3 end_force) col;
      draw_cylinder_ex (r3_to_vec3 mid_force) (r3_to_vec3 end_force) (norme f/.50.0)
    end
  in
  begin_drawing ();
  clear_background Color.black;
  begin_mode_3d st.cam;
  (* repère du 0*)
  draw_cube zero_r 1.0 1.0 1.0 Color.raywhite;
  draw_grid 100 10.0;
  if not (is_key_down Key.Z) then
    (*dessin des triangles*)
    List.iter (fun (a,b,c) -> draw_tri a.pos b.pos c.pos (fade Color.green 0.5)) (surfaces st.blob);
    (*dessin des arrêtes*)
    Graph.iteri
      (fun i s -> let f = bilan_des_forces s i (volume st.blob) st.blob
        {penche = st.penche; points = st.blob; center = center; k_ressort = st.k_ressort
      }
    )
    in
    if is_key_down Key.F then (*juste les forces*)
      List.iter (fun (f,col) -> draw_vec s.pos f col) f
    else if is_key_down Key.G then (*l'accélération*)
      draw_vec s.pos (somme_forces f s) Color.orange;
      begin
        List.iter
          (fun (posb,) ->
            let a = r3_to_vec3 s.pos in
            let b = r3_to_vec3 posb.pos in
            draw_line_3d a b Color.orange
            (Graph.linked_to st.blob i);
          )
      end
    )
    st.blob;
  end_mode_3d ();
  draw_text (
    "F pour le mode forces
    Space pour le \"saut temporel\"
    W pour l'affichage de la surface
    R pour pencher la surface (" ^ string_of_bool st.penche ^ ")
    " ^ string_of_float st.k_ressort ^ "N/m force de ressort élastique, modifiable avec
    h/y")
    10 20 20 Color.raywhite;
  end_drawing ()

```

```
(*boucle principale*)
let rec loop st =
  let center = Graph.fold_left (+$) zero (Graph.map (fun x-> x.pos) st.blob) *$ (1.0
/.(foi n)) in
  if Raylib.window_should_close () then Raylib.close_window () else
(*calcul du prochain état en parallèle du dessin*)
  let integrationDomain = Domain.spawn
    (fun _ -> Integration.integrate
      {points=st.blob;k_ressort = st.k_ressort;penche = st.penche:center = center})
  in
  let time_jumping = is_key_down Key.Space in
  if not (time_jumping && st.t mod 10 <> 0) then draw st center;
  let rmed = r3_to_vec3 center in
  Vector3.set_x (Camera3D.target st.cam) {Vector3.x rmed};
  Vector3.set_y (Camera3D.target st.cam) {Vector3.y rmed};
  Vector3.set_z (Camera3D.target st.cam) {Vector3.z rmed};
  Camera3D.set_projection st.cam CameraProjection.Perspective;
  update_camera (addr st.cam) CameraMode.Third_person;
  let blob' = Domain.join integrationDomain in
  loop {
    t = st.t + 1;
    blob = blob';
    k_ressort = max 0.0 (st.k_ressort +
      1.0 *(
        if is_key_down Key.H then 1.0
        else if is_key_down Key.Y then -.1.0
        else 0.0));
    penche = if is_key_pressed Key.R then not st.penche else st.penche;
    cam = st.cam
  }
}
let setup () =
  Raylib.init_window w h "Blob";
(*Raylib.set_target_fps 60;*)
if is_window_ready () then
  let camera = Camera3D.create zero_r zero_r (Vector3.create 0. 1. 0.) 45. CameraProjection.Perspective
  in let open Camera3D in
  set_position camera (Vector3.create 0. i 20.: 40.);
  set_target camera (Vector3.create 0. i 0.:);
  set_up camera (Vector3.create 0. 1. 0.);
  set_fovy camera 120.;
  set_projection camera CameraProjection.Perspective;
  disable_cursor ();
  {
    t = 0;
    blob = Graph.initial ();
    penche = false;
    k_ressort = k_ressort;
    cam = camera;
  }
else failwith "Problème de fenêtre"
let () =
  setup () |> loop
```