Université Paris 8

Moteurs de Jeu [1]

Nicolas JOUANDEAU
n@up8.edu

septembre 2022

1. A la découverte des fonctions proposées par les moteurs et les jeux possibles.

7 Simulation physique

La principale fonction de la simulation physique est de donner l'illusion d'être dans un monde correspondant au notre ; la simulation physique est avant tout une question de modèle et de calculs ; selon la précision souhaitée, on utilise des différentes approximations du modèle de l'effet considéré.

Les déplacements physiques sont associés à des forces ; dans tous les cas, il souhaitable que la précision de la simulation soit autant indépendante que possible du pas de temps choisi pour la simulation.

7.1 Orbite circulaire d'un satellite dans le plan

En physique Newtonienne, un satellite en rotation circulaire autour d'un point subit une force tangentielle par rapport à son centre de rotation :

- pour un satellite de coordonnées (x, y)
- $d = \sqrt{x^2 + y^2}$ est la distance entre le satellite et le centre de la rotation
- $\overrightarrow{(-y, x)}$ définit le sens de la vitesse tangentielle du satellite
- $\overrightarrow{(\frac{-y}{d}, \frac{x}{d})}$ est la force unitaire correspondant au mouvement du satellite

Selon la méthode d'Euler, on peut appliquer la force unitaire pour trouver une approximation du déplacement du satellite ; le programme suivant permet d'obtenir la trajectoire suivie par le satellite en utilisant la méthode d'Euler ; le résultat est présenté en Fig. 67.

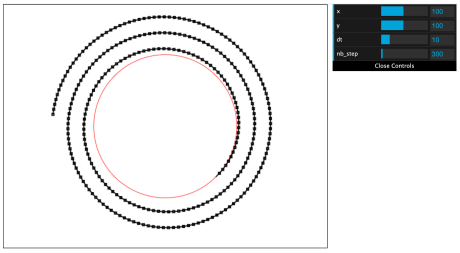

FIG. 67 – Approximation par la méthode d'Euler.

```
class Pt {
  constructor(x, y, dt) {
    this.x = x; this.y = y;
  }
  tan_unit(dt) {
    let ndx = -this.y, ndy = this.x;
    let norm = Math.sqrt(ndx*ndx+ndy*ndy);
    return [dt*ndx/norm, dt*ndy/norm];
  }
  euler_step(dt) {
    let dec = this.tan_unit(dt);
    this.x += dec[0];
    this.y += dec[1];
  }
}
```

```
class Orbit {
  constructor(x, y, dt, nb_step) {
    this.pos = new Pt(x,y);
    this.all_pos = [new Pt(x,y)];
    this.dt = dt;
    this.nb_step = nb_step;
    for (let i = 0; i < nb_step; i++) {
      this.pos.euler_step(dt);
      this.all_pos.push(new Pt(this.pos.x, this.pos.y));
    }
  }
}
let cnv = document.getElementById("myCanvas");
let ctx = cnv.getContext("2d");
let param = {
  x : 100, y : 100, dt : 10, nb_step : 300,
};
let gui = new dat.gui.GUI();
gui.add(param, 'x').min(10).max(200).onChange(draw);
gui.add(param, 'y').min(10).max(200).onChange(draw);
gui.add(param, 'dt').min(1).max(50).onChange(draw);
gui.add(param, 'nb_step').min(10).max(10000).onChange(draw);
let Ox = cnv.width/2;
let Oy = cnv.height/2;
function draw() {
  let orb = new Orbit(param.x, param.y, param.dt, param.nb_step);
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  ctx.beginPath();
  let dist_origin = Math.sqrt((param.x*param.x)+(param.y*param.y));
  ctx.arc(Ox, Oy, dist_origin, 0, 2*Math.PI);
  ctx.strokeStyle = "red";
  ctx.stroke();
  ctx.closePath();
  for (let i = 1; i < param.nb_step; i++) {
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.rect(Ox+orb.all_pos[i].x-2,Oy+orb.all_pos[i].y-2, 4,4);
    ctx.stroke();
    ctx.closePath();
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.moveTo(Ox+orb.all_pos[i-1].x,Oy+orb.all_pos[i-1].y);
    ctx.lineTo(Ox+orb.all_pos[i].x,Oy+orb.all_pos[i].y);
    ctx.stroke();
    ctx.closePath();
  }
}
draw();
```

On peut résumer le principe de calcul de la méthode d'Euler dans le cas du point de coordonnées (x_i, y_i) à l'instant i, partant de coordonnées (x_0, y_0) à $t = 0$ par

$$\begin{cases} x_{n+1} = x_n + hf(t_n, x_n) \\ y_{n+1} = y_n + hf(t_n, y_n) \end{cases}$$

avec $h_n = t_{n+1} - t_n$ le pas de temps et h le pas de l'itération.

Avec la méthode d'approximation du point médian, le système précédent devient

$$\begin{cases} x_{n+\frac{1}{2}} = x_n + \frac{h}{2} f(t_n, x_n) \\ y_{n+\frac{1}{2}} = y_n + \frac{h}{2} f(t_n, y_n) \\ x_{n+1} = x_n + hf(t_n + \frac{h}{2}, x_{n+\frac{1}{2}}) \\ y_{n+1} = y_n + hf(t_n + \frac{h}{2}, y_{n+\frac{1}{2}}) \end{cases}$$

Avec la méthode d'approximation du point médian, on ajoute une fonction `midpoint_step` à la classe `Pt` ; partant de la vitesse tangentielle au point initial, on trouve un premier décallage `dec` ; on applique la moitié de `dec` pour trouver le point médian ; en ce point médian, on peut calculer un décallage `dec2`, qui est appliqué à partir du point initial pour trouver une meilleure approximation de la position finale.

```
class Pt {
  constructor(x, y, dt) {
    this.x = x; this.y = y;
  }
  tan_unit(dt) {
    let ndx = -this.y, ndy = this.x;
    let norm = Math.sqrt(ndx*ndx+ndy*ndy);
    return [dt*ndx/norm, dt*ndy/norm];
  }
  euler_step(dt) {
    let dec = this.tan_unit(dt);
    this.x += dec[0];
    this.y += dec[1];
  }
  midpoint_step(dt) {
    let dec = this.tan_unit(dt);
    let midpoint = new Pt(this.x+dec[0]/2, this.y+dec[1]/2);
    let dec2 = midpoint.tan_unit(dt);
    this.x += dec2[0];
    this.y += dec2[1];
  }
}
```

```
let cnv = document.getElementById("myCanvas");
let ctx = cnv.getContext("2d");
let Ox = cnv.width/2, Oy = cnv.height/2;
let dt = 30;
let euler_pt = new Pt(100,100);
let euler_pos = [new Pt(euler_pt.x,euler_pt.y)];
let midpoint_pt = new Pt(100,100);
let midpoint_pos = [new Pt(midpoint_pt.x,midpoint_pt.y)];
for (let i = 0; i < 40; i++) {
  euler_pt.euler_step(dt);
  euler_pos.push(new Pt(euler_pt.x,euler_pt.y));
  midpoint_pt.midpoint_step(dt);
  midpoint_pos.push(new Pt(midpoint_pt.x,midpoint_pt.y));
}
let iteration = 40;
function draw() {
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  ctx.font = '24pt Calibri';
  ctx.fillStyle = 'black';
  ctx.fillText(iteration.toString(), cnv.width-100, cnv.height-30);
  for (let i = 1; i < euler_pos.length; i++) {
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "lightgray";
    ctx.moveTo(Ox+euler_pos[i-1].x,Oy+euler_pos[i-1].y);
    ctx.lineTo(Ox+euler_pos[i].x,Oy+euler_pos[i].y);
    ctx.stroke();
    ctx.closePath();
  }
  ctx.beginPath();
  ctx.arc(Ox, Oy, Math.sqrt(20000), 0, 2*Math.PI);
  ctx.strokeStyle = "red";
  ctx.stroke();
  ctx.closePath();
  for (let i = 1; i < midpoint_pos.length; i++) {
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.rect(Ox+midpoint_pos[i].x-2,Oy+midpoint_pos[i].y-2, 4,4);
    ctx.stroke();
    ctx.closePath();
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.moveTo(Ox+midpoint_pos[i-1].x,Oy+midpoint_pos[i-1].y);
    ctx.lineTo(Ox+midpoint_pos[i].x,Oy+midpoint_pos[i].y);
    ctx.stroke();
    ctx.closePath();
  }
}
```

```
73  function update_pos() {
74    iteration += 1;
75    if(euler_pos.length < 500) {
76      euler_pt.euler_step(dt);
77      euler_pos.push(new Pt(euler_pt.x,euler_pt.y));
78    }
79    midpoint_pt.midpoint_step(dt);
80    midpoint_pos.push(new Pt(midpoint_pt.x,midpoint_pt.y));
81    midpoint_pos.splice(0,1)
82  }
83  function update(timestamp) {
84    update_pos();
85    draw();
86    requestAnimationFrame(update);
87  }
88  requestAnimationFrame(update);
```

Fig 68 présente le résultat de l'exécution du programme précédent à l'itération 3113 avec une valeur `dt` à 30 ; la trajectoire théorique est présentée en rouge ; l'approximation par la méthode d'Euler est présentée en gris et celle par la méthode du point médian en noire.

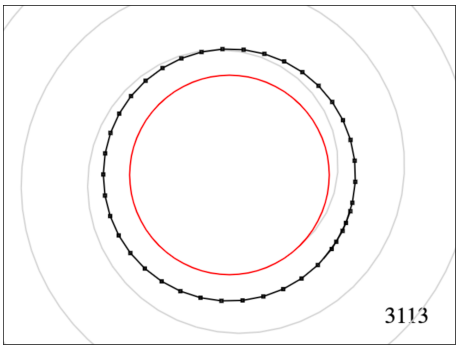

FIG. 68 – Approximation par la méthode du point médian.

Pour résumer et plus simplement, pour $y' = f(t, y)$ avec $y(t_0) = y_0$ on a :

$$\begin{cases} y_{n+1} = y_n + h.k_2 \\ k_1 = f(t_n, y_n) \\ k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \end{cases}$$

avec k_1 la pente à l'instant n et k_2 la pente au point médian de l'intervalle.

La méthode de Runge-Kutta généralise les deux méthodes d'Euler et du point médian, qui sont respectivement d'ordre 1 et 2 ; à l'ordre 4, on :

$$\begin{cases} y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k4) \\ k_1 = f(t_n, y_n) \\ k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\ k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\ k_4 = f(t_n + h, y_n + h.k_3) \end{cases}$$

Avec la méthode d'approximation de Runge-Kutta à l'ordre 4, on ajoute une fonction `rk4_step` à la classe `Pt` pour trouver une meilleure approximation de la position au temps $n + 1$.

```
class Pt {
  constructor(x, y, dt) { ... }
  tan_unit(dt) { ... }
  euler_step(dt) { ... }
  midpoint_step(dt) { ... }
  rk4_step(dt) {
    let dec = this.tan_unit(dt);
    let pt2 = new Pt(this.x+dec[0]/2, this.y+dec[1]/2);
    let dec2 = pt2.tan_unit(dt);
    let pt3 = new Pt(this.x+dec2[0]/2, this.y+dec2[1]/2);
    let dec3 = pt3.tan_unit(dt);
    let pt4 = new Pt(this.x+dec3[0], this.y+dec3[1]);
    let dec4 = pt4.tan_unit(dt);
    this.x += (dec[0] + 2*dec2[0] + 2*dec3[0] + dec4[0])/6;
    this.y += (dec[1] + 2*dec2[1] + 2*dec3[1] + dec4[1])/6;
  }
}
```

Fig 69 présente le résultat de l'exécution en utilisant la fonction `rk4_step` à l'itération 6411 avec une valeur `dt` à 30 ; la trajectoire théorique est présentée en rouge ; l'approximation par la méthode du point médian est présentée en gris et celle par la méthode de Runge-Kutta à l'ordre 4 en noire.

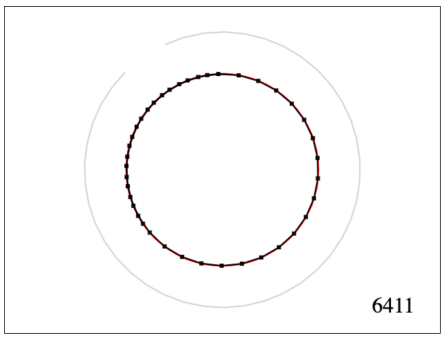

FIG. 69 – Approximation par la méthode de Runge-Kutta à l'ordre 4.

7.2 Trajectoire d'un projectile

Etant soumis à la pesanteur, la trajectoire d'un projectile en (x_0, y_0), de vitesse initiale v_0 et d'angle θ_0 est définie par :

$$\begin{cases} dx_{n+1} = v_n \times cos(\theta_n) \\ dy_{n+1} = v_n \times sin(\theta_n) - g/2 \\ x_{n+1} = x_n + dx_{n+1} \\ y_{n+1} = y_n + dy_{n+1} \\ v_{n+1} = ||\overrightarrow{v_n}|| \\ \theta_{n+1} = arctan(\frac{dy_{n+1}}{dx_{n+1}}) \end{cases}$$

Le projectile est ici sans forces de frottement, soumis uniquement à la force initiale et à son poids avec une constante de pesanteur g de 9.81 $m.s^{-2}$.

On définie la classe `Particule` pour suivre le mouvement d'un tel projectile ; la fonction `next` permet de calculer sa position suivante.

```
function deg2rad(d) {
  return Math.PI*d/180;
}
class Pt {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}
class Particule extends Pt {
  constructor(x, y, v0, theta) {
    super(x,y);
    this.v = v0;
    this.theta = theta;
  }
  next(dt) {
    let dx = this.v*Math.cos(this.theta);
    let dy = this.v*Math.sin(this.theta)-9.81/2;
    let nv = Math.sqrt(dx*dx+dy*dy);
    let ntheta = Math.atan2(dy,dx);
    this.x += dx*dt/nv;
    this.y += dy*dt/nv;
    this.v = nv;
    this.theta = ntheta;
  }
}
```

La fonction `compute_positions` calcule l'ensemble des positions de la trajectoire recherchée ; le nombre d'itérations est borné par `max_step` ; si $y < 0$, le calcule des positions s'arrête.

```
let param = {
  ox : 100,
  oy : cnv.height-100,
  v0 : 200,
  theta : 80,
  dt : 10,
  max_step : 1000,
};
let s, positions;
function compute_positions() {
  s = new Particule(0,0,param.v0,deg2rad(param.theta));
  positions = [new Particule(s.x,s.y,s.v,s.theta)];
  for (let i = 1; i < param.max_step; i++) {
    s.next(param.dt)
    if(s.y < 0) break;
    positions.push(new Particule(s.x,s.y,s.v,s.theta));
  }
}
function compute_and_draw() {
  compute_positions();
  ctx.clearRect(0, 0, cnv.width, cnv.height);
  ctx.beginPath();
  ctx.lineWidth = 1;
  ctx.strokeStyle = "red";
  ctx.moveTo(param.ox,param.oy);
  ctx.lineTo(cnv.width,param.oy);
  ctx.stroke();
  ctx.closePath();
  for (let i = 1; i < positions.length; i++) {
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.rect(param.ox+positions[i].x-2,param.oy-positions[i].y-2, 4,4);
    ctx.stroke();
    ctx.closePath();
    ctx.beginPath();
    ctx.lineWidth = 2;
    ctx.strokeStyle = "black";
    ctx.moveTo(param.ox+positions[i-1].x,param.oy-positions[i-1].y);
    ctx.lineTo(param.ox+positions[i].x,param.oy-positions[i].y);
    ctx.stroke();
    ctx.closePath();
  }
}
compute_and_draw();
```

Fig 70 présente le résultat pour une vitesse initiale de 200 et un angle initial de 80 degrés.

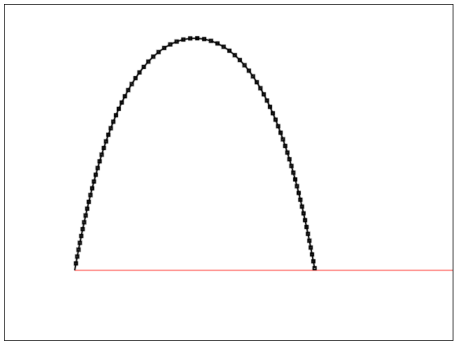

FIG. 70 – Trajectoire résultante.

Pour une vitesse initiale de 45 et un pas de temps de 50, la dernière position du projectile avec $y > 0$ est très au-dessus du sol, comme présenté en Fig 70 ; en l'absence de rebond, on peut ajouter un calcul particulier pour définir la position finale au sol :

1. avec la position au point d'impact
2. avec la translation en surface de la dernière position en collision
3. avec la translation en surface de la dernière position sans collision
4. avec la position en collision avec le sol à distance dt de la dernière position sans collision

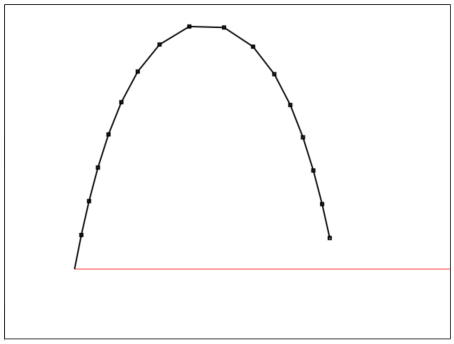

FIG. 71 – Autre trajectoire résultante.

Dans les cas 1. et 4., les calculs de simulation physique réutilisent des fonctions de collision présentées dans les sections 4.4 et 4.5 :

— la position au point d'impact se calcule par intersection entre le dernier segment et le sol
— la position en collision à distance dt se calcule par intersection entre le cercle de rayon dt centré sur la dernière position sans collision et le sol

Le cas 2. est probablement le plus simple et le moins évident à voir comme imperfection ; si on déforme le dernier segment avant contact avec le sol, il importe de ne pas propager la déformation après rebond.

7.3 Rebond d'un projectile sur un obstacle

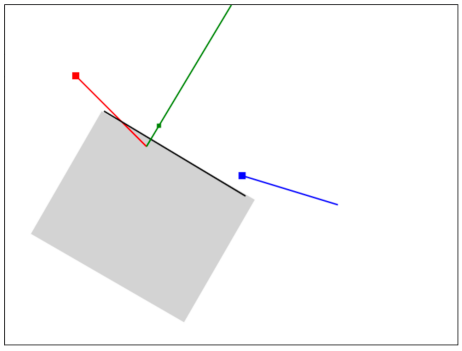

FIG. 72 – Points clés d'un rebond.

Fig 72 présente les points clés possibles d'un calcul de rebond d'un projectile sur un obstacle ; on appelle $\vec{v}$ le vecteur initial entrant en collision avec l'obstacle, v_i le point initial de $\vec{v}$ et v_f le point final de $\vec{v}$; la normale à la surface en v_f est représentée en vert ; on définit p le projeté orthogonal de v_i sur cette normale ; pour un vecteur $\vec{r}$ correspondant au rebond, r_i le point de départ du rebond est le symétrique de v_i par p ; le vecteur $\overrightarrow{v_f r_i}$ correspond au vecteur $\vec{r}$ et permet de définir r_f.