Le Problème à deux corps

LEFEVRE Florian - MPSI 4

TIPE 2010/2011 :

Mouvement, mobilité.

Sommaire :

Introduction :

Le problème à deux corps fut un des problèmes centraux de la mécanique Newtonienne, et de la mécanique céleste. Il désigne l'étude des mouvements relatifs d'un système de deux points matériels soumis à la seule force d'interaction gravitationnelle. Historiquement, Newton énonce le résultat dans *Philosophiae Naturalis Principia Mathematica* (Principes mathématiques de philosophie naturelle) – en 1687 / 1726.

Comment simuler informatiquement les trajectoires relatives de deux corps en interaction gravitationnelle ?

Nous verrons que ce problème se réduit à l'étude d'un mouvement à force centrale que nous résoudrons de manière numérique. Nous souhaitons d'ores et déjà avoir un résultat visuel sous la forme d'une animation. On testera la précision du programme sur des cas réels bien connus.

Note de programmation :

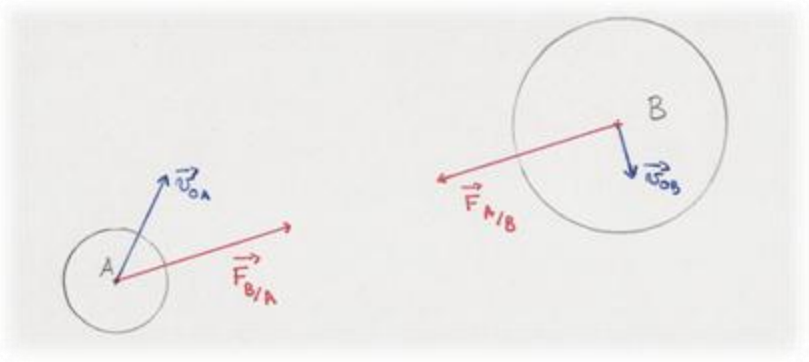
Tous les programmes sont écrits avec la bibliothèque flottant ouvert, les opérations sur les entiers seront redéfinies. Lorsque l'utilisation de la fenêtre graphique sera nécessaire, la bibliothèque graphique sera également ouverte.

```
#open "float";;
#open "graphics";;
```

I Problème et mise en équation

A° Présentation du problème et modélisation

1) Modélisation



On modélise le problème à deux corps par un système ∑ de deux points matériels A et B de masses respectives m_a et m_b :

$$\sum = \{ A(m_a) ; B(m_b) \}$$

On se place dans un référentiel Galiléen R_g (d'origine O).

Le système est soumis aux seules forces d'interactions gravitationnelles (supposées conservatives) :

$$\overrightarrow{F_{A/B}} \; et \; \overrightarrow{F_{B/A}}$$

Le système est considéré comme isolé ainsi :

$$\overrightarrow{Fext_{/A}} = 0 \; et \; \overrightarrow{Fext_{/B}} = 0$$

2) Caractère Galiléen de R^*

a) Equation dans R_g

Théorème du centre de masse :

$$m_\Sigma \cdot \overrightarrow{aG_{/R_g}} = \sum \overrightarrow{F_{ext}} = \vec{0}$$

$$\Rightarrow \overrightarrow{p\Sigma_{/R_g}} = m_\Sigma \cdot \overrightarrow{vG_{/R_g}} \; est \; conservée$$

$$\Rightarrow \overrightarrow{vG_{/R_g}} = \overrightarrow{C^{st}}$$

R^* est donc en translation rectiligne uniforme par rapport à R_g, donc R^* est galiléen.

On fera donc l'étude dans R^*.

Théorème du Moment cinétique : Par rapport à O fixe dans R_g :

$$\frac{d\overrightarrow{L_O}(\Sigma)_{R_g}}{dt} = \sum \overrightarrow{\mathcal{M}_O(\overrightarrow{F_{ext}})} = \vec{0}$$

Car Σ est isolé donc $\overrightarrow{L_O}(\Sigma)_{R_g} = \overrightarrow{C^{st}}$.

Théorème de l'énergie mécanique :

$$\frac{dEm_{\Sigma/R_g}}{dt} = P_{\overrightarrow{F_{non\ cons.}}} = 0$$

Car les forces intérieures sont conservatives donc $Em_{\Sigma/R_g} = C^{ste}$.

$$\overrightarrow{F_{A/B}} = -Gr \cdot \frac{m_A m_B}{\left\|\overrightarrow{AB}\right\|^3} \cdot \overrightarrow{AB}$$

b) Etude dans le référentiel barycentrique

On considère toujours le système

$$\sum = \{ A(m_a) ; B(m_b) \}$$

On fera, cette fois-ci, l'étude dans le référentiel $R^* Galiléen\ (d'origine G)$.

Le système est toujours soumis aux seules forces d'interactions gravitationnelles (supposées conservatives) :

$$\begin{cases} \overrightarrow{F_{A/B}} = -\overrightarrow{F_{B/A}} \\ \overrightarrow{Fext_{/A}} = \overrightarrow{Fext_{/B}} = 0 \end{cases}$$

On cherche à trouver les trajectoires barycentriques des corps A et B : $\overrightarrow{GA}(t)\ et\ \overrightarrow{GB}(t)$.

On Rappelle : $m_A\overrightarrow{GA} + m_B\overrightarrow{GB} = \vec{0}$

c) Lois de conservation dans R^*

Moment cinétique barycentrique :

Théorème du moment cinétique par rapport à G fixe dans R^*

$$\frac{d\overrightarrow{L(\Sigma)^*}}{dt} = \overrightarrow{\mathcal{M}_G(\overrightarrow{F_{ext}})} = \vec{0}$$

Donc $\overrightarrow{L(\Sigma)^*}$ se conserve au cours du temps (il est fixé par les conditions initiales).

Energie mécanique barycentrique :

Théorème de l'énergie mécanique dans R^*Galiléen

$$\frac{dEm(\Sigma)^*}{dt} = \sum \mathcal{P}_{F_{nc}} = 0$$

Donc $Em(\Sigma)^*$ se conserve au cours du temps. L'énergie mécanique est aussi fixée par les conditions initiales.

Avec :

$Em(\Sigma)^* = Ec(\Sigma)^* + Ep_{int}$; Ep_{int} telle que $\overrightarrow{F_{A/B}} = -\frac{dEp_{int}}{dr}$

Remarque : La quantité de mouvement est toujours conservée : $\overrightarrow{p_\Sigma^*} = \vec{0}$.

3) Mobile réduit : Particule fictive M

Soit Mun point fictif tel que $\overrightarrow{GM} = \overrightarrow{AB}$

a) Relation entre les coordonnées de A, B, et M, dans R^*

$$\begin{cases} \overrightarrow{GM} = \overrightarrow{AB} = \overrightarrow{GB} - \overrightarrow{GA} \\ m_A\overrightarrow{GA} + m_B\overrightarrow{GB} = \vec{0} \end{cases}$$

Donc :

$$\begin{cases} m_A\overrightarrow{GA} + m_B(\overrightarrow{GA} + \overrightarrow{AB}) = \vec{0} \\ m_A\left(\overrightarrow{GB} + \overrightarrow{BA}\right) + m_B\overrightarrow{GB} = \vec{0} \end{cases}$$

On a alors :

$$\overrightarrow{GA} = -\frac{m_B}{m_A + m_B}\overrightarrow{GM}$$

$$\overrightarrow{GB} = \frac{m_A}{m_A + m_B}\overrightarrow{GM}$$

b) Relation entre les vitesses de $\overrightarrow{v_A^*}$, $\overrightarrow{v_B^*}$, et $\overrightarrow{v_M^*}$, dans R^*

$$\overrightarrow{v_A^*} = \frac{d\overrightarrow{GA}}{dt} \; et \; \overrightarrow{v_B^*} = \frac{d\overrightarrow{GB}}{dt}$$

$$\overrightarrow{v_M^*} = \frac{d\overrightarrow{GM}}{dt} = \frac{d(\overrightarrow{GB} - \overrightarrow{GA})}{dt} = \overrightarrow{v_B^*} - \overrightarrow{v_A^*}$$

D'où :

$$\overrightarrow{v_A^*} = -\frac{m_B}{m_A + m_B}\overrightarrow{v_M^*}$$

$$\overrightarrow{v_B^*} = \frac{m_A}{m_A + m_B}\overrightarrow{v_M^*}$$

Si on connaît $\overrightarrow{GM}(t)$, on a $\overrightarrow{GA}(t) et \; \overrightarrow{GB}(t)$ par homothétie.

B° Mise en équation

Réduction du problème à deux corps à un problème mono-corps.

On applique le Principe Fondamental de La Dynamique dans le référentiel barycentrique Galiléen :

PFD sur A :

$$(1) : m_A \cdot \frac{d^2\overrightarrow{GA}}{dt^2} = \overrightarrow{F_{B/A}} = -\overrightarrow{F_{A/B}}$$

PFD sur B :

$$(2) : m_B \cdot \frac{d^2\overrightarrow{GB}}{dt^2} = \overrightarrow{F_{A/B}}$$

Réduction canonique :

$$\frac{(2)}{m_B} - \frac{(1)}{m_A} : \frac{d^2(\overrightarrow{GB} - \overrightarrow{GA})}{dt^2} = \overrightarrow{F_{A/B}} \cdot \frac{m_A + m_B}{m_A m_B}$$

$$\Leftrightarrow \frac{m_A m_B}{m_A + m_B} \cdot \frac{d^2(\overrightarrow{GM})}{dt^2} = \overrightarrow{F_{A/B}}$$

$$\Leftrightarrow \mu \cdot \overrightarrow{a_{M^*}} = \overrightarrow{F_{A/B}} \; avec \; \mu = \frac{m_A m_B}{m_A + m_B} \; la \; masse \; réduite \; du \; système$$

Tout se passe comme si M était un point matériel de masse $\mu = \frac{m_A m_B}{m_A + m_B}$ soumis à une force $\overrightarrow{F_{A/B}}$.

C'est un problème à force centrale. On peut donc trouver $\overrightarrow{GM}(t)$ et en déduire par homothétie les trajectoires barycentriques de A et de B.

On retiendra pour notre programme l'équation différentielle qui sera résolue par une méthode numérique :

$$\Leftrightarrow \frac{d^2(\overrightarrow{GM})}{dt^2} = \overrightarrow{F_{A/B}} \cdot \frac{m_A + m_B}{m_A m_B}$$

Et la relation qui donne la force d'interaction entre A et B à tout instant :

$$\overrightarrow{F_{A/B}} = \frac{m_A m_B}{m_A + m_B} \cdot \frac{d^2(\overrightarrow{GM})}{dt^2}$$

Nous avons, désormais, toutes les clés pour créer un algorithme traduisant la résolution de ce problème.

C° Etude complémentaire

Moment cinétique et énergie dans R^*

1) Moment cinétique barycentrique

$$\begin{aligned}\overrightarrow{L(\Sigma)^*} &= \overrightarrow{GA} \wedge m_A\overrightarrow{v_A^*} + \overrightarrow{GB} \wedge m_B\overrightarrow{v_B^*} \\ &= -\frac{m_B}{m_A + m_B}\overrightarrow{GM} \wedge m_A\overrightarrow{v_A^*} + \frac{m_A}{m_A + m_B}\overrightarrow{GM} \wedge m_B\overrightarrow{v_B^*} \\ &= \overrightarrow{GM} \wedge \mu(\overrightarrow{v_B^*} - \overrightarrow{v_A^*}) \\ &= \overrightarrow{GM} \wedge \mu\overrightarrow{v_M^*}\end{aligned}$$

On a alors :

$$\overrightarrow{L(\Sigma)^*} = \overrightarrow{L(\mathrm{M})^*} = \overrightarrow{GM} \wedge \mu\overrightarrow{v_M^*} = \overrightarrow{C^{st}}$$

M a donc une trajectoire plane. La représentation graphique de notre simulation se fera donc en 2D.

2) Energies cinétique et mécanique barycentriques

a) Energie cinétique

$$Ec_\Sigma^* = \frac{1}{2}m_A\overrightarrow{v_A^*}^2 + \frac{1}{2}m_B\overrightarrow{v_B^*}^2$$

$$Ec_\Sigma^* = \frac{1}{2}m_A\frac{{m_B}^2}{(m_A + m_B)^2}\overrightarrow{v_M^*}^2 + \frac{1}{2}m_B\frac{{m_A}^2}{(m_A + m_B)^2}\overrightarrow{v_M^*}^2$$

$$Ec_\Sigma^* = \frac{1}{2}\frac{m_A m_B}{(m_A + m_B)^2}(m_A + m_B)\overrightarrow{v_M^*}^2$$

$$Ec_\Sigma^* = Ec_M^*$$

b) Energie mécanique

$Em_\Sigma^* = Ec_\Sigma^* + Ep_{int}$ et $Em_\mathrm{M}^* = Ec_M^* + Ep_M$

Avec $\overrightarrow{F_{A/B}} = -\frac{dEp_{int}}{dr} = -\frac{dEp_M}{dr}$

D'où : $Em_\Sigma^* = Em_M^*$

Ces résultats pourront être utilisés dans la gestion des collisions.

II Résolution de l'équation différentielle

Avant-propos :

Un système de deux points matériels isolés est le seul cas soluble analytiquement, i.e. on aurait pu obtenir une solution exacte de l'équation différentielle, précédemment, établie grâce, par exemple, aux formules de Binet ou encore à l'aide de la méthode du vecteur excentricité.

On obtient alors l'équation d'une conique de la forme :

$$r(\theta) = \frac{p}{1 + e.\cos(\theta)}$$

Qui donne, rappelons le, une trajectoire :

- Circulaire dans le cas ou $e = 0$.
- Elliptique dans le cas ou $0 < e < 1$.
- Parabolique dans le cas ou $e = 1$.
- Hyperbolique dans le cas ou $e > 1$.

Nous avons, cependant, choisi de résoudre numériquement l'équation différentielle vérifiée par le système. Ce choix semble être la méthode la plus facile pour résoudre ce problème à l'aide d'une machine.

A° Méthode d'Euler

Avant-propos :

Nous avons décidé de commencer par utiliser la méthode d'Euler, la plus simple des méthodes de résolution numérique des équations différentielles. Facilement compréhensible, elle est, de surcroit, aisément programmable.

1) Méthode à l'ordre 1

a) Description

Soit y une fonction dérivable sur I à valeur dans $\mathbb{R}$.

Soit f une fonction de $I \times \mathbb{R} \to \mathbb{R}$

Soit $(E_{1)}$ une équation différentielle ordinaire (EDO) d'ordre 1 définie par :

$$(E_1) : y'(x) = f(x, y(x))$$

On a (car y est dérivable sur I) :

$$\forall x \in I, \lim_{h \to 0} \frac{y(x+h) - y(x)}{h} = y'(x)$$

Dans le cas d'un petit pas d'itération ($h \to 0$), On peut alors approximer $y(x+h)$ tel que :

$$y(x+h) \approx y'(x) \times h + y(x)$$

On note ainsi :

$$y_{n+1} \approx y'_n \times h + y_n$$

$$\Leftrightarrow (1) : y_{n+1} \approx f(x, y_n) \times h + y_n$$

Où n représente la n^{ième} image calculée.

On obtient ainsi une méthode itérative pour calculer les antécédents successifs et donc une méthode programmable à condition que l'état initial du système soit connu :

$$E_{i_1} : y(x_0) = y_0$$

b) Implémentation de la méthode d'Euler à l'ordre 1

Une implémentation « naïve » et récursive se déduit immédiatement de la relation (1) qui retourne l'ensemble des images sous la forme d'une liste de type $float * float\ list$: $[(x_0,y_0); (x_1,y_1); \ldots; (x_n,y_n)]$

```
 (*Euler 1 - Récursif*)

(*Euler1 : EDO1 -> y(x initial) -> x initial -> x finale -> pas d'itération*)

let rec euler1 f' yo xo xf dx =
  let y1 = yo + f'(xo,yo) * dx in
  match xo with
  |xo when xo < xf  -> (xo,yo)::(euler1 f' y1 (xo + dx) xf dx);
  |_                -> []
;;
```

c) Test, limites et améliorations

On pourra tester cette fonction sur l'exemple suivant :

```
 (*Décharge d'un condensateur (U(t))*)

let RC = 80.;;

let f'(x, y) =
- ( 1. / RC )* y;;

(*1*) euler1 f' 6. 0. 10. 0.001;;

(*2*) euler1 f' 6. 0. 400. 0.001;;
```

Explication :

On se propose de résoudre l'équation différentielle vérifiée par la tension lors de la décharge d'un condensateur :

On sait que le régime permanent est atteint au bout de 5τ avec $\tau = R \cdot C$. On souhaite obtenir une courbe complète de la décharge du condensateur jusqu'à ce que le régime permanent soit atteint, soit $xf = 400$ dans notre exemple `(*2*)`. On fixe le pas d'itération à $dx = 0.001$. On devrait donc obtenir une liste de $400\ 001$ couples. L'exception Out_of_memory est levée.

En effet, le nombre d'« itérations » d'une boucle récursive maximum est de $104\,796$. Bien que la taille d'une liste ne soit, à priori, pas limitée (si ce n'est par la mémoire), dans toute la suite de la partie sur la résolution numérique, on décidera de programmer de façon itérative. On utilisera des vecteurs, qui se prêtent mieux à ce type de programmation, dont la taille maximal est de $sys\_\max\_vect\_length() = 4\,194\,303$, soit un gain de près de 4 Millions de d'enregistrements.

On programme alors une version itérative qui retournera une matrice de type $float\ vect\ vect$.

```
 (*Euler 1 - Itératif*)

(*Euler1 : EDO1 -> y(x initial) -> x initial -> x finale -> pas d'itération*)

let euler1 f' yo xo xf dx =
  let i = ref xo and ii = ref 0
    and vect = (create_vect xo xf dx) in
      let y = ref yo in
          while !i <= xf do
          (vect.(!ii).(0) <- !i);
          (vect.(!ii).(1) <- !y);
          y := !y + f'(!i,!y) * dx;
          i := !i + dx;
          incr ii;
        done;
      vect;;
```

Où *create_vect* est une fonction qui crée un tableau de taille adéquate.

```
 (*Création d'une matrice de taille adaptée*)
let create_vect xo xf dx = make_matrix (int_of_float((xf-xo)/dx + 1./dx +
dx)) 2 0.;;
```

Représentation de la solution :

Reprenons d'abord l'exemple traité précédemment avec la fonction itérative associée à la fonction de représentation graphique *graph* fournie et commentée en annexe.

```
(*Décharge d'un condensateur (U(t))*)

let RC = 80.;;

let f'(x, y) =
- ( 1. / RC )* y;;
```

On définit les échelles identiquement paramétrées. Les courbes représentatives seront différenciées par leurs couleurs. Respectivement rouge et vert pour la solution numérique et analytique.

La solution analytique est évidement connue :

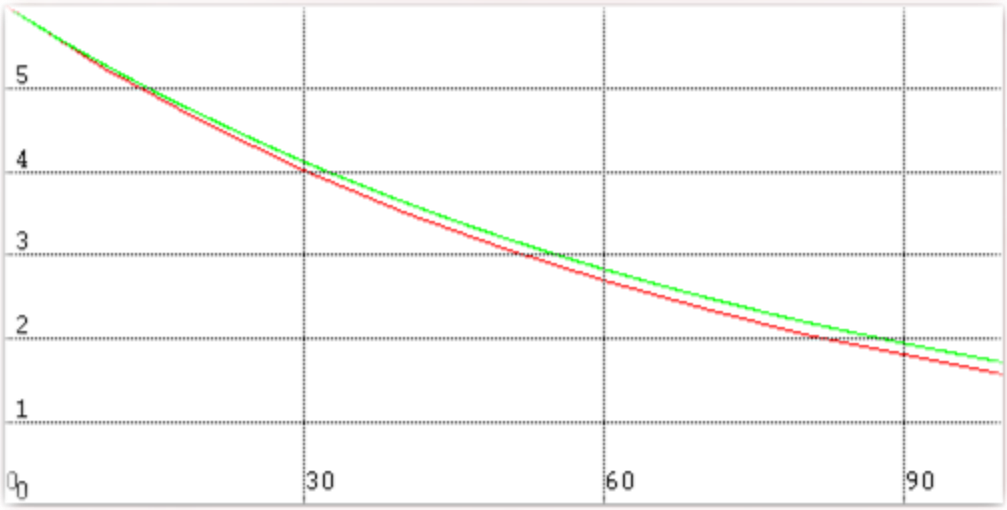
$$U(t) = 6 \times e^{t/\tau} avec\, \tau = R \cdot C$$

```
let numerique =  {scallingAuto = false ; quad = true; lineMode = true ;
xmin = 0. ; xmax = 100. ; ymin = 0. ; ymax = 6.; color = red};;

let analytique = {scallingAuto = false ; quad = true; lineMode = false ;
xmin = 0. ; xmax = 100. ; ymin = 0. ; ymax = 6.; color = green};;
```

On choisit volontairement un pas d'itération élevé ($10\,secondes$), travaillant sur un intervalle relativement cours, les erreurs sont peux visibles pour un pas de l'ordre de 10^{-2}.

```
graph (euler1 f' 6. (0.) 400. 10.) 500 250 30. 1. numerique;;

graph (fonction_to_vect (fun t -> 6. * exp(-t/RC)) 1. 400. 0.1) 500 250 30.
1. analytique;;
```

Représentation graphique de la résolution numérique de l'EDO par $euler1$ (rouge) en comparaison avec la solution analytique (vert).



On constate que la méthode d'Euler fournit une assez bonne approximation dans le cas présent. Cependant, la différence entre la courbe représentative de la solution analytique et numérique croît avec le temps.

2) Adaptation pour une EDO d'ordre 2

a) Description

Soit y une fonction deux fois dérivable sur I à valeur dans $\mathbb{R}$.

Soit f une fonction de $I \times \mathbb{R}^2 \to \mathbb{R}$

Soit (E_2) une équation différentielle ordinaire (EDO) d'ordre 2 défini par :

$$(E_2) : y''(x) = f(x, y(x), y'(x))$$

On a, ici, besoin de connaître deux conditions initiales pour résoudre (E_2) :

$$E_{i_2} : \begin{cases} y(x_0) = y_0 \\ y'(x_0) = y'_0 \end{cases}$$

Pour calculer y''_{n+1} , on a donc besoin de y_{n+1} et y'_{n+1} .

Calcul de y_{n+1} :

Comme dans la résolution d'une EDO d'ordre 1, on obtient :

$$y_{n+1} \approx y'_n \times h + y_n$$

Calcul de y'_{n+1} :

On obtient de même :

$$y'_{n+1} \approx y''_n \times h + y'_n$$

Calcul de y''_{n+1} :

$$y''_{n+1} \approx f(x+h, y_{n+1}, y'_{n+1})$$

Une fois ces trois relations établis, on peut boucler de nouveau pour obtenir y_{n+2} :

$$y_{n+2} \approx y'_{n+1} \times h + y_{n+1}$$

$$y'_{n+2} \approx y''_{n+1} \times h + y'_{n+1}$$

$$y''_{n+2} \approx f(x+2h, y_{n+2}, y'_{n+2})$$

b) <u>Implémentation</u>

```
 (*Euler 2*)

(*Euler2 : EDO2 -> y'(x initial)-> y(x initial) -> x initial -> x finale -> pas
d'itération*)

let euler2 f'' y'o yo xo xf dx =
  let y'' = ref (f''xo,yo,y'o) and y' = ref y'o and y = ref yo in
    let i = ref xo and ii = ref 0
    and vect = (create_vect xo xf dx) in
      while !i <= xf do
        (vect.(!ii).(0) <- !i);
        (vect.(!ii).(1) <- !y);
        y   := !y + dx * ( !y' + ( !y'' * dx));
        y'  := !y' + !y'' * dx;
        y'' := f''(!i, !y, !y');
        i := !i + dx;
        incr ii;
      done;
vect;;
```

c) Test et limites

On se propose de résoudre l'équation différentielle vérifiée par un système masse-ressort, soumis à des forces de frottements fluides.

$$\ddot{x} + \frac{\alpha}{m}\dot{x} + \frac{k}{m}x = 0$$

Avec :

- α le coefficient de frottement fluide.
- k la constante de raideur du ressort.
- m la masse du système.

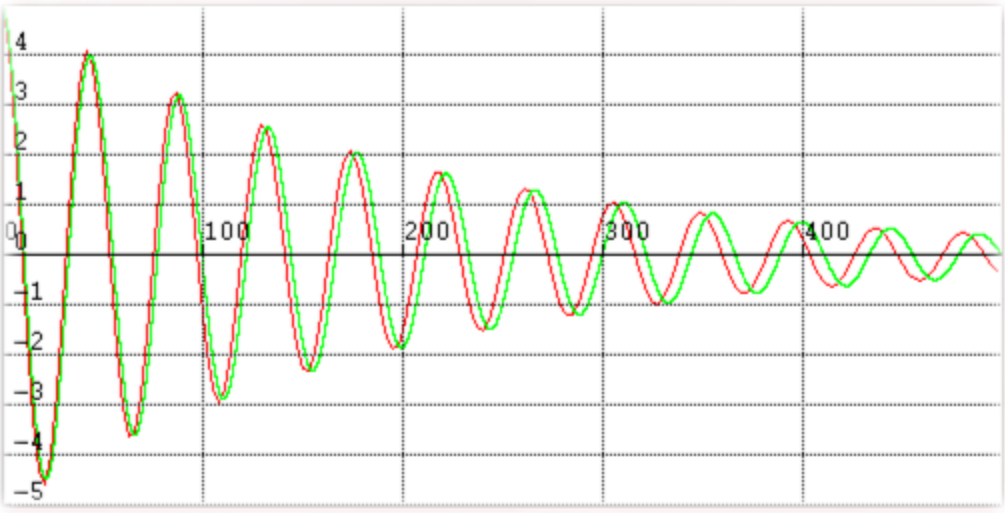
On connaît la solution analytique :

$$x(t) = e^{\frac{-a}{m}t}(5 \cdot \cos(\frac{\sqrt{-\Delta}}{2} \cdot t)) \; avec \; \Delta = \frac{a^2}{m^2} - 4 \cdot \frac{k}{m}$$

```
(*Système masse ressort avec frottement fluide*)

let numerique = {scallingAuto = false ; quad = true; lineMode = true ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = red};;
let analytique = {scallingAuto = false ; quad = true; lineMode = false ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = green};;

let a = 5. and m = 500. and k = 10.;;

let solution t =
let racineDelta = sqrt( 4. * k / m - (a ^^^^ 2.) / (m ^^^^ 2.)) in
exp( - a * t / (2.*m))*(5.*cos(racineDelta * t /2. ));;


graph (euler2 f''1 0. 5. 0. 500. 3.) 500 250 100. 1. numerique;;
graph (fonction_to_vect solution 0. 500. 0.01) 500 250 100. 1. analytique;;
```

On trace comme, précédemment, en rouge la solution numérique et en vert la solution analytique. Le pas est fixé à 3 $secondes$. Un pas de l'ordre de 10 donne un résultat inexploitable.

Courbes représentatives de la solution numérique obtenue par $euler2$ (rouge) et analytique (vert) :



On constate une différence qui ne cesse de croître entre la courbe « analytique » et l'approximation. En effet, l'erreur commise est cumulative. Nous devons, d'ores-et-déjà, envisager une méthode plus précise pour pouvoir approximer nos équations différentielles sur des intervalles de tailles plus importantes.

3) Etude de la complexité et taux d'erreur

a) Complexité

On décide de majorer la complexité des programmes en un nombre d'opération élémentaire. Ainsi les opérations $(+, -, \times, \div)$, l'assignation $:=$ ou $\leftarrow$, l'incrémentation d'une boucle for, la vérification d'un booléen dans une boucle $while$, la vérification d'une égalité ou inégalité booléenne $(<, \leq, =, \geq, >)$, la lecture d'une case d'une matrice, d'un vecteur… comptent pour une opération élémentaire. La fonction $creat\_vect$ compte aussi pour une opération élémentaire dans la mesure où elle sera aussi présente et ce, le même nombre de fois, dans la fonction associée à la méthode de Runge-Kutta.

Plus généralement, bien que cela soit discutable, toute fonction de complexité constante sera comptée comme une opération élémentaire.

Dans le but d'une comparaison future avec la complexité de la méthode de Runge-Kutta pour une EDO d'ordre 2, on s'intéresse ici à la complexité de la fonction $euler2$.

Complexité de $euler2$:

$$T(n) = 19n + 9$$

Soit une complexité en $O(19n)$ ou encore en $O(n)$, où n représente le nombre d'itération. C'est donc une complexité linéaire .

b) Erreur commise

Grâce à la formule de Taylor-Young on obtient :

$$f(x+h) = f(x) + f'(x) \cdot h + \underset{h\to 0}{o}(h)$$

En prenant :

$$f(x+h) \approx f(x) + f'(x) \cdot h$$

On commet donc une erreur en $o_{h\to 0}(h)$. La méthode d'Euler est, en effet, une méthode d'ordre 1 (approximation par une tangente.).

B° Runge-Kutta 4 (RK4)

Avant-propos :

On se tourne vers une méthode de résolution numérique plus précise. On fait le choix d'utiliser la méthode de Runge Kutta d'ordre 4, qui offre une erreur en $o(h^4)$. Elle est facilement programmable et a, comme la méthode d'Euler, une complexité linéaire.

1) Méthode de Runge-Kutta 4 pour une EDO d'ordre 1

a) Méthode

Soit y une fonction dérivable sur I à valeur dans $\mathbb{R}$.

Soit f une fonction de $I \times \mathbb{R} \to \mathbb{R}$

Soit (E_3) une équation différentielle ordinaire (EDO) d'ordre 1 définie par :

$$(E_3) : y'(x) = f(x, y(x))$$

La méthode RK4 est donnée par l'équation :

$$y_{n+1} = y_n + \frac{h}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4)$$

Où

$$k_1 = f(x_n, y_n)$$

$$k_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_1)$$

$$k_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2} \cdot k_2)$$

$$k_4 = f(x_n + h, y_n + h \cdot k_3)$$

L'idée est que la valeur suivante y_{n+1} est approchée par la somme de la valeur actuelle y_n et du produit de la pente estimée par le pas d'itération h. La pente est obtenue par une moyenne pondérée des pentes suivantes :

- k_1 est la pente au début de l'intervalle.
- k_2 est la pente au milieu de l'intervalle, en utilisant la pente k_1 pour calculer la valeur de y au point $\left(x_n + \frac{h}{2}\right)$par le biais de la méthode d'Euler.
- k_3 est de nouveau la pente au milieu de l'intervalle, mais obtenue cette fois en utilisant la pente k_2 pour calculer y.
- k_4 est la pente à la fin de l'intervalle, avec la valeur de y calculée en utilisant k_3.

Dans la moyenne des quatre pentes, un poids plus grand est donné aux pentes des points du milieu.

$$pente_{moyenne} = \frac{k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4}{6}$$

b) Implémentation

La programmation de la méthode de Runge Kutta pour une EDO d'ordre 1 se déduit directement des formules ci-dessus.

Code :

```
(* Runge Kutta 4 Equation différentielle Ordre 1*)

(*RK4_1 : EDO1 -> y(x initial) -> x initial -> x finale -> pas d'itération*)

let RK4_1 f' yo xo xf dx =
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0. in
  let x = ref xo and y = ref yo in
  let xx = ref 0 and vect = (create_vect xo xf dx) in
    while !x <= xf do
      (vect.(!xx).(0) <- !x);
      (vect.(!xx).(1) <- !y);
      K1 := f'(!x,!y);
      K2 := f'(!x + 0.5 * dx , !y + 0.5 * !K1 * dx);
      K3 := f'(!x + 0.5 * dx , !y + 0.5 * !K2 * dx);
      K4 := f'(!x + 0.5 , !y + !K3 * dx);
      y := !y + (!K1 + 2. * !K2 + 2. * !K3 + !K4) * dx / 6. ;
      x := !x + dx;
      incr xx;
    done;
vect;;
```

c) Test et limites

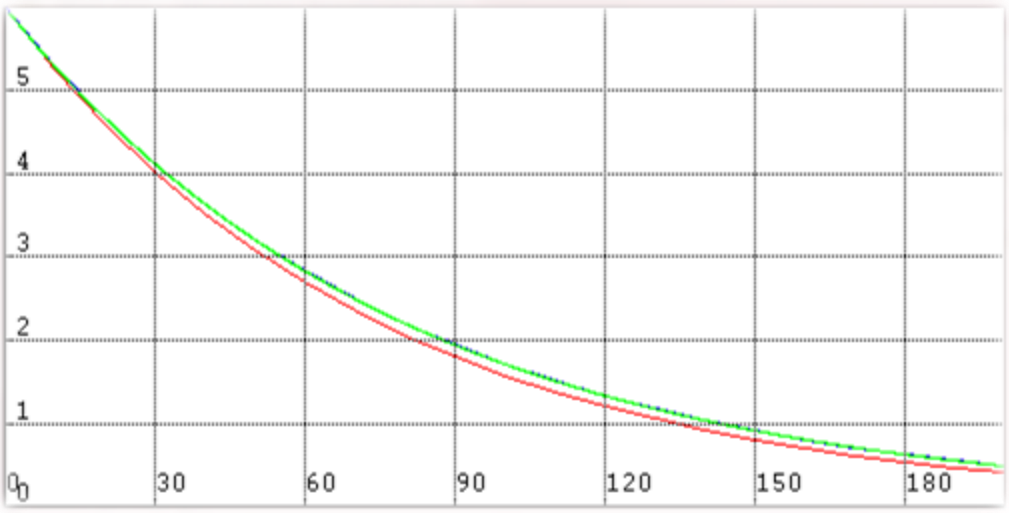
Nous testerons la méthode de Runge Kutta sur le même exemple que pour la méthode d'Euler pour des EDO d'ordre 1. Nous comparerons les résultats avec œux obtenus par la méthode d'Euler et la solution analytique.

```
(*décharge d'un condensateur (U(x))*)

let RC = 80.;;

let f'(x,y) =
- ( 1. / RC )* y;;
```

La solution analytique sera affichée en vert, celle obtenue à l'aide de la méthode de Runge Kutta en Bleu et celle obtenue par la méthode d'Euler en rouge. Les pas seront fixés pour les deux méthodes à 10, de sorte de pouvoir comparer les résultats obtenus.

```
let numerique = {scallingAuto = false ; quad = true; lineMode = true ;
xmin = 0. ; xmax = 200. ; ymin = 0. ; ymax = 6.; color = red};;
let numerique2 = {scallingAuto = false ; quad = true; lineMode = true ;
xmin = 0. ; xmax = 200. ; ymin = 0. ; ymax = 6.; color = blue};;
let analytique = {scallingAuto = false ; quad = true; lineMode = false ;
xmin = 0. ; xmax = 200. ; ymin = 0. ; ymax = 6.; color = green};;

graph (euler1 f' 6. (0.) 400. 10.) 500 250 30. 1. numerique;;
graph (RK4_1 f' 6.0 0. 400. 10.) 500 250 30. 1. numerique2;;
graph (fonction_to_vect (fun t -> 6. * exp(-t/RC)) 1. 400. 0.1) 500 250 30.
1. analytique;;
```

Courbes représentatives de la solution numérique obtenue par $RK4\_1$ (bleu), $euler1$ (rouge) et la solution analytique (vert) :



On constate que la courbe représentative obtenue par la méthode de Runge Kutta 4 est quasi-confondue avec celle de la solution analytique. Les résultats obtenus sont donc satisfaisants, et ce même pour un pas d'itération assez grand.

2) Méthode de Runge-Kutta 4 pour une EDO d'ordre 2

a) Méthode

Soit y une fonction deux fois dérivable sur I à valeur dans $\mathbb{R}$.

Soit f une fonction de $I \times \mathbb{R} \to \mathbb{R}$

Soit (E_4) une équation différentielle ordinaire (EDO) d'ordre 2 définie par :

$$(E_4) : y''(x) = f(x, y(x), y'(x))$$

Le schéma de la méthode de Runge Kutta pour une EDO d'ordre deux est donné par :

$$y_{n+1} = y_n + \frac{(j_1 + 2j_2 + 2j_3 + j_4)}{6}$$

$$y'_{n+1} = y'_n + \frac{(k_1 + 2k_2 + 2k_3 + k_4)}{6}$$

Où

$j_1 = y'_n \cdot h$	$k_1 = f(x_n, y_n, y'_n) \cdot h$
$j_2 = (y'_n + \frac{k_1}{2}) \cdot h$	$k_2 = f\left(x_n, y_n + \frac{j_1}{2}, y'_n + \frac{k_1}{2}\right) \cdot h$
$j_3 = (y'_n + \frac{k_2}{2}) \cdot h$	$k_3 = f\left(x_n, y_n + \frac{j_2}{2}, y'_n + \frac{k_2}{2}\right) \cdot h$
$j_4 = (y'_n + k_3) \cdot h$	$k_4 = f(x_n, y_n + j_3, y'_n + k_3) \cdot h$

b) Implémentation

La programmation de la méthode de Runge Kutta Pour une EDO d'ordre 2 se déduit directement des formules ci-dessus.

```
 (* Runge Kutta 4 Equation différentielle d'ordre 2*)

(*RK4_2 : EDO2 -> y'(x initial) -> y(x initial) -> x initial -> x finale -> pas
d'itération*)

let RK4_2 f'' y'o yo xo xf dx =
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0.
  and J1 = ref 0. and J2 = ref 0. and J3 = ref 0. and J4 = ref 0.
  and x = ref xo and y = ref yo and y' = ref y'o in
  let xx = ref 0 and vect = (create_vect xo xf dx) in
    while !x <= xf do
      (vect.(!xx).(0) <- !x);
      (vect.(!xx).(1) <- !y);
      J1 := !y'*dx;
      K1 := f''(!x, !y, !y')*dx;
      J2 := (!y' + !K1 / 2.)*dx;
      K2 := f''(!x,( !y + !J1 / 2.),(!y' + !K1 / 2.))*dx;
      J3 := (!y' + !K2 / 2.)*dx;
      K3 := f''(!x, (!y + !J2 / 2.), (!y' + !K2 / 2.))*dx;
      J4 := (!y' + !K3)*dx;
      K4 := f''(!x, (!y + !J3), (!y' + !K3))*dx;
      y' := !y' + (!K1 + 2. * !K2 + 2. * !K3 + !K4)/ 6. ;
      y  := !y  + (!J1 + 2. * !J2 + 2. * !J3 + !J4)/ 6. ;
      x  := !x + dx;
      incr xx;
    done;
vect;;
```

c) Test et limites

Nous testerons cette adaptation au EDO d'ordre 2 sur le même exemple que pour la méthode d'Euler. Nous comparerons les résultats avec ceux obtenus par la méthode d'Euler et la solution analytique.

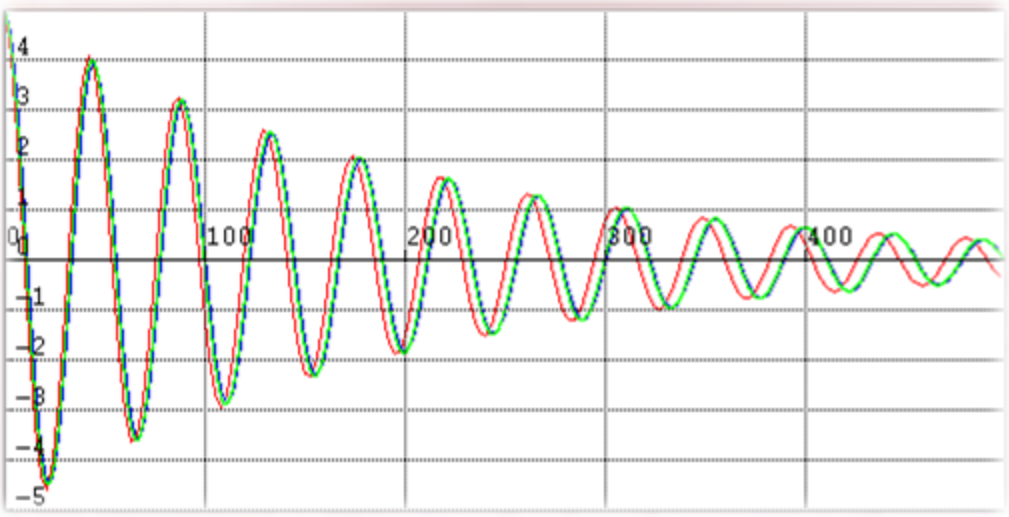
```
(*Système masse ressort avec frottement fluide*)

(*a représente le coefficient de frottement fluide, k la constante de
raideur du ressort.*)

let a = 5. and m = 500. and k = 10.;;

let f''1(x,y,y') =
  - ( a / m ) * y' - ( k / m ) * y ;;
```

La solution analytique sera affichée en vert, celle obtenue à l'aide de la méthode de Runge Kutta en Bleu et celle obtenue par la méthode d'Euler en rouge. Les pas seront fixés pour les deux méthodes à 10, de sorte de pouvoir comparer les résultats obtenus. On définit donc les échelles en conséquence, et on programme la solution analytique de l'équation différentielle :

```
let numerique = {scallingAuto = false ; quad= true; lineMode = true ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = red};;

let analytique = {scallingAuto = false ; quad = true; lineMode = false ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = green};;

let numerique2 = {scallingAuto = false ; quad= true; lineMode = true ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = blue};;

let solution t =
let racineDelta = sqrt( 4. * k / m - (a ^^^^ 2.) / (m ^^^^ 2.)) in
exp( - a * t / (2.*m))*(5.*cos(racineDelta * t /2. ));;
```

```
graph (euler2 f''1 0. 5. 0. 500. 3.) 500 250 100. 1. numerique;;
graph (RK4_2 f''1 0. 5. 0. 500. 3.) 500 250 100. 1. numerique2;;
graph (fonction_to_vect solution 0. 500. 0.01) 500 250 100. 1. analytique;;
```

Courbes représentatives de la solution numérique obtenue par $RK4\_2$ (bleu), $euler2$ (rouge) et la solution analytique (vert) :



On constate encore que la courbe représentative de la solution analytique est quasiment confondue avec celle obtenue par la méthode de Runge Kutta. C'est donc une méthode qui semble convenable pour résoudre notre problème.

3) Etude de la complexité et taux d'erreur

La complexité de $RK4\_2$ est de $T(n) = 60n + 15$. On a donc une complexité en $O(60n)$ ou encore en $O(n)$, ou n représente le nombre d'itérations. C'est une complexité linéaire du même ordre de grandeur que la complexité de $euler2$. (4 fois supérieure à cette dernière). La force de la méthode de Runge Kutta 4 ne repose donc pas sur sa complexité, mais sur une erreur accumulée moindre face à une résolution par la méthode d'Euler et ce, même pour un pas d'itération plus grand. En effet, la méthode RK4 est une méthode d'ordre 4, ce qui signifie que l'erreur commise à chaque étape est de l'ordre de h^5, alors que l'erreur totale accumulée est de l'ordre de h^4.On aura donc un gain non négligeable en vitesse et en précision.

C° Brèves extensions aux autres méthodes envisageables

Pour un gain de précision : Runge Kutta d'ordre 4 n'est pas la seule méthode possible. Il en existe toute une famille analogue d'ordre 4 et supérieur telles que Runge Kutta Fehlberg (ordre 4 et 5) que l'on a implémenté en CAML pour résoudre une EDO d'ordre 1 (« Runge Kutta F 5.ml »), la méthode de Dormand Prince qui est utilisée par défaut dans MATHLAB ou encore la méthode de Cash Krab. Toute ces méthodes sont plus précises mais de complexité supérieure à Runge Kutta 4.

Pour un gain de Vitesse : des méthodes, moins précises, comme la méthode de Romberg, existent.

III Simulation informatique

Avant-propos :

On se propose de créer un programme qui a pour but de calculer, numériquement, les trajectoires du problème à deux corps, dans tous les cas de figures possibles, (centre de masse fixe ou non). Nous souhaitons avoir une représentation graphique en « temps réel » de l'avancement de la simulation. L'intervalle d'étude ne devra pas être limité dans le temps et pourra à priori être d'une longueur non négligeable. L'utilisation de listes ou de vecteurs devra donc être limitée au maximum ou adaptée de manière à ce que la simulation puisse se faire sur une durée de temps indéterminée.

A° Structure du programme

1) Déclaration des types et opérations

a) Opérateurs et parseurs usuels

Dans tout le programme, nous travaillerons avec la bibliothèque $float$ ouverte. Ce choix se justifie par une majorité d'opérations sur les flottants avec la résolution numérique. Cependant, les fonctions graphiques nécessitent souvent des opérations sur les entiers. On redéfinit donc ces opérations de la façon suivante :

```
(*Opérations sur les entiers*)

  let prefix ++ a b = int_of_float(float_of_int(a) + float_of_int(b)) ;;
  let prefix -- a b = int_of_float(float_of_int(a) - float_of_int(b)) ;;
  let prefix ** a b = int_of_float(float_of_int(a) * float_of_int(b)) ;;
  let prefix ^^ a b = int_of_float( power (float_of_int(a))
(float_of_int(b)));;
  let prefix // a b = int_of_float(float_of_int(a) / float_of_int(b)) ;;
```

Pour éviter toute confusion entre $**$ (l'opérateur de multiplications sur les entiers) et $**.$ (l'opérateur « puissance » prédéfini en CAML), on programme une nouvelle fonction puissance analogue :

```
(*Opération sur les flottants*)
  (* '**.' reste disponible.*)
  let prefix ^^^^ a b = power a b;;
```

L'utilisation de l'indication $\#infix$ pour l'interpréteur CAML-light est obsolète, comme nous utilisons des symboles déjà $infix$.

On met en place un nouvel opérateur de référencement $\sim:=$ qui permet le référencement d'un couple et qui nous sera utile, ultérieurement :

```
(*Référencement d'un couple*)
      let prefix ~:= (a,b) (c,d) = a := c;      b := d;;
```

Pour alléger le code source et sa lisibilité, on utilisera des parseurs usuels :

```
(*Conversions des types*)
  let int(a) = int_of_float a;;
  let float(a) = float_of_int a;;
  let string(a) = string_of_float a;;
```

b) Déclaration des types

On modélisera un corps par un type enregistrement avec les champs : $masse, rayon, coordonnées, vitesses$ de type $float$ pour la majorité. L'utilisation des flottants est justifiée par les grands ordres de grandeurs que le programme pourra avoir à traiter. Tous les « labels » seront $mutables$ de sorte que leurs valeurs puissent évoluer dynamiquement au cours de la simulation.

```
  type corps   =
    {mutable masse    : float;
    mutable rayon     : int;
    mutable coord_x   : float;
    mutable coord_y   : float;
    mutable vitesse_x : float;
    mutable vitesse_y : float;
    };;
```

On définit aussi un type enregistrement $state$ qui retournera en temps « réel » (donc de type $mutable$) l'avancement de la simulation :

```
  type state  =
    {mutable corps1  : corps;
    mutable corps2   : corps;
    mutable partFict : corps;
    mutable centreG  : corps;
    mutable F        : float;
  };;
```

Et un type $options$ assez lourd qui conservera les préférences utilisateurs lors de la simulation :

```
  type options =
    {mutable dt       : float;
    mutable zoom      : float;
    mutable origine_x : int ;
    mutable origine_y : int ;
    mutable scale_F   : float;
    mutable scale_v   : float;
    mutable dispTime  : bool;
    mutable dispState : bool;
    mutable dispTraj  : bool;
    mutable dispM     : bool;
    mutable dispG     : bool;
    mutable dispCorps : bool;
    mutable blinker   : bool;
    mutable hypermode : bool;
    mutable lineto    : bool;
    mutable reset     : bool;
    mutable elag      : bool;
  };;
```

Où :

- dt représente le pas d'itération variable.
- $zoom$ est le facteur de grossissement.
- $origine\_x$ et $origine\_y$ sont les coordonnées de l'origine de la fenêtre graphique.
- $scale\_F$ et $scale\_V$ représentent l'échelle des vecteurs forces et vitesses.
- $disp[\dots]$ sont des booléens activant ou non la représentation graphique de l'état de la simulation, des trajectoires, de la particule fictive, du centre de masse et des corps.
- $blinker$ est un booléen qui caractérise l'activation d'un mode évitant le clignotement de la fenêtre graphique.
- $hypermode$ est un booléen qui activera ou non l'accélération de la simulation.
- $lineto$ est un booléen caractérisant le mode de traçage des trajectoires : continu ou point par point.
- $reset$ et $elag$ sont des options sur des listes qui représenteront les trajectoires des corps.

NB : On définit aussi une exception « $collision$ » qui sera levée en cas de collision des deux corps.

```
(*Exceptions*)
  exception collision;;
```

2) Déclaration des fonctions et variables auxiliaires

a) Fonctions auxiliaires

La bibliothèque $graphics$ sera ouverte dans l'ensemble du programme ainsi que la bibliothèque $format$ qui sert uniquement pour l'installation d'un $printer$ pour les objets du type corps avec la fonction $print\_corps$.

Pour installer la fonction $print\_corp$ de façon valide, on s'est très largement inspiré d'une fonction connue et analogue disponible dans les sources de CAML (« format.ml ») : $print\_int$.

```
let pp_print_int state i = pp_print_string state (string_of_int i);;
let print_int = pp_print_int std_formatter;;
```

Pour alléger la syntaxe, les sauts de ligne seront faits par le caractère d'échappement $'\backslash n'$ et non par la fonction $print\_newline()$.

```
(*Déclaration d'un printer pour le type Corps*)

  let pp_print_corps state c =
    pp_print_string state ("Masse      : "^(string c.masse));
    print_string ("\n Rayon      : "^(string_of_int c.rayon));
    print_string ("\n Coordonées : ("^(string c.coord_x)^", "^(string
c.coord_y)^")");
    print_string ("\n Vitesse    : ("^(string c.vitesse_x)^", "^(string
c.vitesse_y)^")");
  ;;

  let print_corps = pp_print_corps std_formatter;;

      install_printer "print_corps";;
```

On programme des fonctions auxiliaires qui trouveront leur utilité dans la fonction principale :

```
(*Reset key_pressed (Type unit)*)
  let readkey() = let c = ref `a` in c:= read_key();;
```

La fonction $readkey()$ est analogue à la fonction $read\_key()$ mais est de type $unit$. Elle sert principalement à remettre la valeur du booléen $key\_pressed()$ à $false$, pour éviter des bugs dans des structures conditionnelles.

```
(*Pause jusqu'à ce qu'une touche soit pressée*)
  let wait() = while not(key_pressed()) do done; readkey() ;;
```

La fonction $wait()$ met le système en « pause » jusqu'à ce qu'une touche soit pressée, pour cela, on fait exécuter une boucle vide à la machine qui s'arrêtera seulement si l'utilisateur appuie sur une touche.

```
(*Pause de x ms*)
  let pause x = sound 20000 x;;
```

La fonction $pause$ fait une pause de x milliseconde en faisant jouer un son inaudible par le système. Selon les versions de la bibliothèque graphique utilisée ou de l'ordinateur, il est préférable de régler la fréquence à 0 Hertz ($sound\ 0\ x$). Cette commande n'est malheureusement pas reconnue par tous les systèmes. Un léger bourdonnement peut être audible dans le cas contraire.
On aurait, pour ralentir le système, ou faire une pause de x milliseconde, utiliser une boucle vide de type for. Cependant, cette solution n'offre aucun contrôle sur la durée de la pause. Durée qui peut alors varier d'une machine à l'autre selon la puissance du processeur (et de la configuration mémoire du système).

```
(*Auxiliaire - Utilisé dans le test pour les trajectoires.*)
let gt1 a = if (a < 1.) then 1 else int a;;
```

Cette fonction sera utilisée dans une structure conditionnelle pour enregistrer les trajectoires tous les x calculs, si le flottant est inférieur à 1, la fonction renverra 1 et non 0 pour ne pas lever l'exception $Division\_by\_zero$ dans une congruence.

```
(*Elagage d'une liste (renvoie un élément sur deux)*)
let rec elag_list l =
  match l with
  |[]    -> []
  |[a]   -> [a]
  |a::b::q -> a::(elag_list q);;
```

La fonction $elag\_list$ renvoie dans une liste un élément sur deux de la liste donnée en argument (et diminue donc par deux sa taille). Elle sera utile dans la mesure où les trajectoires seront enregistrées dans des listes, de taille toujours plus importante, ralentissant progressivement le programme.

```
(*Opération sur liste*)
let op_list l options =
    match (options.reset, options.elag) with
    |true , _ -> [];
    |_ , true -> (elag_list l);
    |_ ,_     -> l ;;
```

La fonction *op_list* renvoie une liste vide, une liste élaguée ou la liste inchangée selon le souhait de l'utilisateur. Elle sera utilisée pour remettre à zéro l'enregistrement des trajectoires ou accélérer la simulation.

```
(*Diminuer la puissance d'un nombre*)
  let low_number nn max =
    let n = ref nn and exp = ref 0 and coeff = ref 1. in
      while !n > 1e3 do
        n := !n / 10.;
        incr exp;
      done;
      while !n > max do
        n := !n / 1.5;
        coeff := !coeff * 1.5;
      done;
    (!coeff * (10.^^^^(float !exp)));;
```

La fonction *low_number* « diminue » le nombre en le divisant par un certain coefficient de manière à ce qu'il soit inférieur à une limite. Elle sera utilisée pour afficher, suivant une échelle adaptée, les vecteurs forces et les vecteurs vitesses qui ont des ordres de grandeurs, (bien que très variables d'un système à un autre), pouvant atteindre 10^{22}.

```
let Runge_Kutta_4_ordre_2 f''x x x' (y:float) options=
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0.
  and J1 = ref 0. and J2 = ref 0. and J3 = ref 0. and J4 = ref 0. in
(*Calcule de x*)
      J1 := x'* options.dt;
      K1 := f''x(y, x, x')* options.dt;
      J2 := (x' + !K1 / 2.)* options.dt;
      K2 := f''x(y,( x + !J1 / 2.), (x' + !K1 / 2.))* options.dt;
      J3 := (x' + !K2 / 2.)* options.dt;
      K3 := f''x(y, (x + !J2 / 2.), (x' + !K2 / 2.))* options.dt;
      J4 := (x' + !K3)* options.dt;
      K4 := f''x(y, (x + !J3), (x' + !K3))* options.dt;
  (x  + (!J1 + 2. * !J2 + 2. * !J3 + !J4)/ 6.,
   x' + (!K1 + 2. * !K2 + 2. * !K3 + !K4)/ 6.);;
```

La fonction *Runge_Kutta_4_ordre_2* est une adaptation de la fonction de résolution numérique décrite précédemment qui retourne la valeur de son successeur et la dérivée de son successeur.

On remarque que le pas est « *options.dt* » et pourra donc être modifiable au cours de la simulation par l'utilisateur.

b) Fonction d'interactions avec l'utilisateur

La principale fonction d'interactions avec l'utilisateur est définie de la façon suivante et retourne un objet de type *options* et prend pour paramètres d'entrée deux objets de type *options* et *state*.

```
(*Fonction d'interaction clavier.*)
  let waitif options state=
    if key_pressed()
      then
        begin
        match read_key() with
        (*Déplacement*)
        | `z`    -> clear_graph(); (notification infos.(0));
options.origine_y <- options.origine_y ++ 1;options;
        | `q`    -> clear_graph(); (notification infos.(1));
options.origine_x <- options.origine_x -- 1;options;
        | `s`    -> clear_graph(); (notification infos.(2));
options.origine_y <- options.origine_y -- 1;options;
        | `d`    -> clear_graph(); (notification infos.(3));
options.origine_x <- options.origine_x ++ 1;options;
        (*Zoom + et Zoom -*)
        | `-`    -> clear_graph(); (notification infos.(4)); options.zoom <-
options.zoom / 1.2;options;
        | `+`    -> clear_graph(); (notification infos.(5)); options.zoom <-
options.zoom * 1.2;options;
        (*Vitesse de simulation*)
        | `r`    -> (notification infos.(6)); options.dt <- options.dt * 2.;
options;
        | `l`    -> (notification infos.(7)); options.dt <- options.dt / 2.;
options;
        (*Informations*)
        | `i`    -> clear_graph(); (notification infos.(8));
options.dispState <- not(options.dispState); options;
        (*Affichage de la Particule Fictive*)
        | `m`    -> clear_graph(); (notification infos.(9)); options.dispM
<- not(options.dispM) ;options;
        (*Affichage du Centre de Masse G*)
        | `g`    -> clear_graph(); (notification infos.(10)); options.dispG
<- not(options.dispG) ;options;
        (*Affichage des trajectoires*)
        | `a`    -> clear_graph(); (notification infos.(11));
options.dispTraj <- not(options.dispTraj); options;
        (*Mode "Lineto" sur les trajectoires*)
        | `c`    -> clear_graph(); (notification infos.(18)); options.lineto
<- not(options.lineto); options;
        (*Elagage des listes*)
        | `e`    -> clear_graph(); (notification infos.(19)); options.elag
<- true; options;
        (*Reset listes*)
        | `x`    -> clear_graph(); (notification infos.(20)); options.reset
<- true; options;
        (*Affichage du temps*)
        | `t`    -> clear_graph(); (notification infos.(12));
options.dispTime <- not(options.dispTime ); options;
        (*Sens de la simulation*)
        | `>`    -> (notification infos.(13)); options.dt <-  abs_float
options.dt; options;
        | `<`    -> (notification infos.(14)); options.dt <- - abs_float
options.dt; options;
        (* « Anti-blinker » - réduit le clignotement*)
```

```
        | `b`    -> (notification infos.(15)); options.blinker <-
not(options.blinker); options;
        (*Hypermode*)
        | `h`    -> clear_graph(); (notification infos.(16));
options.hypermode <- not(options.hypermode);  options;
        (*Exit*)
        | `\027`-> close_graph(); raise Exit; options;
        (*Pause*)
        |   _    -> (notification infos.(17)); draw_state
(vectString_of_state state options); wait(); clear_graph(); options;
       end
     else options;;
```

La fonction *waitif* repose sur un filtrage des combinaisons possibles. Dans le cas ou aucune touche n'est pressée, la fonction retourne directement l'objet de type *options* donné en argument. Les différentes options sont décrites en annexe (Annexe 6). On pourra se plaindre, de la nécessité de renvoyer, dans chaque cas du filtrage, l'objet modifié de type *options*. Cependant, aucune autre solution convenable n'a été trouvée.

NB :

- '\027' est le code ASCII de la touche Echap.
- Le filtrage est sensible à la case.

Où :

notification est une fonction d'affichage d'informations décrite ultérieurement.

A chaque touche pressée est associé un message de notification informant l'utilisateur sur l'action qu'il vient d'effectuer :

```
(*Messages d'informations*)
  let infos =
    [|
    (*0*) [|"Origine décalée : ";" Y = y + 1"|];
    (*1*) [|"Origine décalée : ";" X = x - 1"|];
    (*2*) [|"Origine décalée : ";" Y = y - 1"|];
    (*3*) [|"Origine décalée : ";" X = x + 1"|];
    (*4*) [|"Zoom - : (divisé par 1.2)";""|];
    (*5*) [|"Zoom + : (multiplié par 1.2)";""|];
    (*6*) [|"Vitesse de simulation doublée :";"Attention les erreurs
s'accumulent."|];
    (*7*) [|"Vitesse de simulation réduite :";"Attention les erreurs
s'accumulent."|];
    (*8*) [|"Affichage de l'état de la simulation :";"Ralentit la
simulation."|];
    (*9*) [|"Affichage de la particule fictive :";"Purement théorique."|];
    (*10*)[|"Affichage de centre de masse :";"Purement théorique."|];
    (*11*)[|"Affichage des trajectoires :";"Ralentit progressivement la
simulation."|];
    (*12*)[|"Affichage du temps : ";"Unité : Seconde."|];
    (*13*)[|"Sens normal de simulation :";"Attention les erreurs
s'accumulent."|];
```

```
    (*14*)[|"Sens inverse de simulation :";"Attention les erreurs
s'accumulent."|];
    (*15*)[|"Mode Anti-Blinker : Ralentit";"Grandement la vitesse de
simulation."|];
    (*16*)[|"Hypermode, désactive les options :";"Accélère la
simulation."|];
    (*17*)[|"En Pause : ";"Appuyez sur une touche pour continuer"|];
    (*18*)[|"Affichage des trajectoires: ";"Mode continu"|];
    (*19*)[|"Trajectoires (listes): ";"Perte d'une information sur
deux."|];
    (*20*)[|"Trajectoires (listes): ";"Remise à zéro"|];
```

c) Fonctions graphique

Avant-propos : Nous utiliserons la bibliothèque graphique d'origine (disponible dans toutes les distributions de CAML-Light). La nécessité d'une bibliothèque graphique conforme aux normes imposées par l'INRIA sera abordée dans la partie difficultés rencontrées. Elle n'est en rien indispensable et sera uniquement utilisée dans une fonction obsolète nommée $bang$.

On redéfinit quelques couleurs de base pour des questions de rendu visuel:

```
(*Couleurs*)
  let marron  = (rgb 135 085 065);;
  let rouge   = (rgb 200 000 000);;
  let bleu    = (rgb 000 000 200);;
  let vert    = (rgb 000 200 000);;
  let c_oie   = (rgb 219 219 000);;
```

Ou c_oie désigne la couleur « caca d'oie ».

La fenêtre graphique sera fractionnée en quatre zones :

<table>
<tr><td rowspan="3">(1) Zone de Simulation (Affichage des corps)</td><td>(2) Temps de la simulation</td></tr>
<tr><td>(3) Etat de la simulation</td></tr>
<tr><td>(4) Zone de notifications</td></tr>
</table>

a) Fonction d'affichage de la simulation (Zone 1)

- Fonction d'affichage des Corps

On programme une fonction $drawCorp$ qui affichera un corps ainsi que son vecteur force (force d'interaction) et son vecteur vitesse. $drawCorp : corps \rightarrow couleur \rightarrow force_x \rightarrow force_y \rightarrow options \rightarrow unit$.

Il ne faut pas oublier de prendre en considération :

- L'origine du repère modifiable par l'utilisateur
- Le facteur de grossissement (Zoom)
- L'échelle de représentation des vecteurs forces et vitesses

```
  let drawCorp c color fx fy options=
    set_color color;
    (*cercle*)
    fill_circle (int (c.coord_x * options.zoom) ++ options.origine_x)
                (int (c.coord_y * options.zoom) ++ options.origine_y)
                (int (float c.rayon * options.zoom));
    (*Vecteur Vitesse*)
    set_color red;
    moveto (int (c.coord_x * options.zoom) ++ options.origine_x)
           (int (c.coord_y * options.zoom) ++ options.origine_y);
    lineto (int ((c.vitesse_x / options.scale_v + c.coord_x) *
options.zoom) ++ options.origine_x)
           (int ((c.vitesse_y / options.scale_v + c.coord_y) *
options.zoom) ++ options.origine_y);
    (*Vecteur Force*)
    set_color c_oie;
    moveto (int (c.coord_x * options.zoom) ++ options.origine_x)
           (int (c.coord_y * options.zoom) ++ options.origine_y);
    lineto (int ((fx / options.scale_F + c.coord_x) * options.zoom) ++
options.origine_x)
           (int ((fy / options.scale_F + c.coord_y) * options.zoom) ++
options.origine_y);
  ;;
```

On peut noter l'utilité d'avoir défini des opérateurs pour les entiers ainsi que des parseurs usuels qui rendent le code moins surchargé et donc plus lisible.

- Affichage des trajectoires

On définit, au préalable, une fonction *plotc* analogue à *plot* pour tracer un couple de flottant avec les options utilisateur. On doit prendre en considération le mode « *lineto* » (affichage des trajectoires en continu) pour cela, la fonction *plotc* a besoin de deux couples de flottants pour être en mesure de tracer une ligne entre ces deux points.

La fonction *plotc* sera donc définie de la façon suivante :
$plotc : point\ précédent \rightarrow point\ à\ tracer \rightarrow options \rightarrow unit.$

Voici son implémentation en CAML :

```
(*Fonction plot couple de flottants*)
  let plotc (a,b) (x,y) options =
  if (options.lineto)
  then
    begin
    moveto (int (a * options.zoom) ++ options.origine_x)
           (int (b * options.zoom) ++ options.origine_y);
    lineto (int (x * options.zoom) ++ options.origine_x)
           (int (y * options.zoom) ++ options.origine_y)
    end
  else
    begin
    plot (int (x * options.zoom) ++ options.origine_x)
         (int (y * options.zoom) ++ options.origine_y)
    end;;
```

Les trajectoires sont enregistrées dans des listes de type $floats * floats\ list$. On a donc besoin d'une fonction $plot\_list$ capable de tracer ces trajectoires à l'aide de la fonction $plotc$ définie précédemment.

```
(*Fonction plot liste*)
  let plotlist l color options=
  set_color color;
  let rec aux l options =
    match l with
      |[]    -> ()
      |[a]   -> ()
      | a::b::q -> (plotc a b options ; aux q options)
     in aux l options;;
```

On définit, localement, une fonction auxiliaire récursive qui lira le contenu de la liste.

- <u>Nettoyage de la zone de simulation</u>

Différentes possibilités étaient envisageables pour nettoyer la zone de simulation :

- Utilisation de la fonction $clear\_graph$
- Utilisation de la fonction $fill\_rect$ sur la zone 1

Cependant, l'utilisation de ces fonctions ralentissait considérablement le programme.

Nous avons, donc, décidé d'effacer uniquement ce qui avait été tracé à l'itération précédente. Ce choix accélère, par un ordre de grandeur de 10 fois, le programme en comparaison à l'utilisation de la fonction $clear\_graph$. (Les trajectoires n'ont pas besoin d'être effacées. Seuls les corps doivent être effacés).

```
  let effaceCorp c fx fy options=
    (*cercle*)
    set_color white;
    fill_circle (int (c.coord_x * options.zoom) ++ options.origine_x)
                (int (c.coord_y * options.zoom) ++ options.origine_y)
                (int (float c.rayon *  options.zoom));
    (*Vecteur Vitesse*)
    moveto (int (c.coord_x * options.zoom) ++ options.origine_x)
           (int (c.coord_y * options.zoom) ++ options.origine_y);
    lineto (int ((c.vitesse_x / options.scale_v + c.coord_x) *
options.zoom) ++ options.origine_x)
           (int ((c.vitesse_y / options.scale_v + c.coord_y) *
options.zoom) ++ options.origine_y);
    (*Vecteur Force*)
    moveto (int (c.coord_x  * options.zoom) ++ options.origine_x)
           (int (c.coord_y  * options.zoom) ++ options.origine_y);
    lineto (int ((fx / options.scale_F + c.coord_x) * options.zoom) ++
options.origine_x)
           (int ((fy / options.scale_F + c.coord_y) * options.zoom) ++
options.origine_y);
     ;;
```

La fonction $effaceCorp$ est donc en tout point analogue et aurait pu être envisagée, à quelques variantes près comme $drawCorp\ c\ white\ fx\ fy\ options$.

b) Fonction d'information sur l'état de la simulation (Zone 2, 3, 4)

- Zone 2 (Affichage du temps de la simulation)

Le temps de la simulation est affiché en *seconde* (unité S.I).

La fonction d'affichage est la suivante :

```
(*Fonction d'affichage du temps*)
  let draw_time time =
    (*Nettoyage de la zone d'écriture*)
    set_color white;
    fill_rect (size_x()-- 300) (size_y()-- 20) 300 20;
    (*Ecriture*)
    set_color black;
    moveto (size_x()-- 300) (size_y()-- 20);
    draw_string ("Temps  : "^(string_of_float time)^" s.");
  ;;
```

- Zone 3 (Affichage de l'état de la simulation)

On définit une fonction d'affichage de type $string\ vect \rightarrow unit$ pour afficher l'état de la simulation.

```
(*Affichage de l'état de la simulation*)
let draw_state states =
    let n = vect_length states -- 1 in
    (*Nettoyage de la zone d'écriture*)
      set_color white;
      fill_rect (size_x()-- 300) (40) 300 (size_y()-- 60);
    (*Ecriture*)
      set_color black;
      for i=0 to n do
        moveto (size_x()-- 300) (size_y() -- (15**i ++ 35)) ;
        draw_string states.(i);
      done;
;;
```

Où :

States, est l'état de la simulation, extraite de l'objet de type *states* préalablement, transformée en type *string vect* à l'aide de la fonction suivante :

```
(*Affichage de l'état de la simulation*)
  let vectString_of_state state options =
  [|
    "Vitesse de simulation : "^(string options.dt);
    "Zoom : x"^(string options.zoom);
    "Echelle Vitesse : 1/"^(string (options.scale_v))^" ème";
    "Echelle Force :   1/"^(string (options.scale_F))^" ème";
    "";
    "Corps 1 :";
    "Xa   : "^(string state.corps1.coord_x);
    "Ya   : "^(string state.corps1.coord_y);
    "Vxa  : "^(string state.corps1.vitesse_x)^" m/s";
    "Vya  : "^(string state.corps1.vitesse_y)^" m/s";
```

```
    "|Va| : "^(string (sqrt((state.corps1.vitesse_x^^^^2.) +
(state.corps1.vitesse_y^^^^2.))))^" m/s";
    "";
    "Corps 2 :";
    "Xb   : "^(string state.corps2.coord_x);
    "Yb   : "^(string state.corps2.coord_y);
    "Vxb  : "^(string state.corps2.vitesse_x)^" m/s";
    "Vyb  : "^(string state.corps2.vitesse_y)^" m/s";
    "|Vb| : "^(string (sqrt((state.corps2.vitesse_x^^^^2.) +
(state.corps2.vitesse_y^^^^2.))))^" m/s";
    "";
    "Force d'interaction (De 1 sur 2) :";
    "F    :"^(string (state.F))^" N";
    "";
    "Centre de Masse G";
    "Xg   : "^(string state.centreG.coord_x);
    "Yg   : "^(string state.centreG.coord_y);
    "Vxg  : "^(string state.centreG.vitesse_x)^" m/s";
    "Vyg  : "^(string state.centreG.vitesse_y)^" m/s";
    "|Vg| : "^(string (sqrt((state.centreG.vitesse_x^^^^2.) +
(state.centreG.vitesse_y^^^^2.))))^" m/s";
    "";
    "Particule Fictive M";
    "Xm   : "^(string state.partFict.coord_x);
    "Ym   : "^(string state.partFict.coord_y);
    "Vxm  : "^(string state.partFict.vitesse_x)^" m/s";
    "Vym  : "^(string state.partFict.vitesse_y)^" m/s";
    "|Vm| : "^(string (sqrt((state.partFict.vitesse_x^^^^2.) +
(state.partFict.vitesse_y^^^^2.))))^" m/s";
  |];;
```

(Toutes les informations sur l'état de la simulation seront données en flottant dans les unités du système international sans soucis du nombre de chiffres significatifs).

- Zone 4 (Zone de notifications)

La fonction *notification* est la fonction décrite ci-dessus (Partie b). Elle affiche sur deux lignes les informations relatives à l'interaction entre la simulation et l'utilisateur dans la zone 4.

```
(*Fonction d'affichage des messages de notifications.*)
  let notification message =
    (*Nettoyage de la zone d'écriture*)
    set_color white;
    fill_rect (size_x()-- 300) 0 300 40;
    (*Ecriture*)
    set_color black;
    (*Ligne 1*)
    moveto (size_x()-- 300) 20 ;
    draw_string message.(0);
    (*Ligne 2*)
    moveto (size_x()-- 300) 0 ;
    draw_string message.(1);
  ;;
```

3) Partie Principale

On doit implanter un algorithme permettant de représenter l'évolution du système.

La source de la partie principale sera fragmentée dans cette partie pour faciliter son explication. Elle est disponible dans son intégralité en annexe.

Nous nommerons cette fonction $solve$, qui prendra comme argument deux objets de type $corps$ (respectivement pour le corps 1 et le corps 2) ainsi que le pas d'itération initial dt de type $float$:

$$solve : corps \rightarrow corps \rightarrow pas\ d'itération.$$

```
(*Fonction solve() - variables : Cas Général : Mode Continu*)
let solve corpsA corpsB dtt =
```

Un certains nombres de définitions locales s'impose, notamment pour le centre de masse G, la particule fictive M et la constante de Gravité Gr indispensable dans l'équation différentielle établie précédemment. G et M seront définis comme des corps (de rayons non nuls pour faciliter leur représentation même s'ils n'ont aucun sens physique). L'implémentation se déduit immédiatement des relations démontrées en première partie.

```
(*Constante de Gravitation Universelle*)
  let Gr = 6.67e-11 in
(*Centre de Masse G*)
  let xG =  (corpsA.masse * corpsA.coord_x + corpsB.masse * corpsB.coord_x)
/ (corpsA.masse + corpsB.masse) in
  let yG =  (corpsA.masse * corpsA.coord_y + corpsB.masse * corpsB.coord_y)
/ (corpsA.masse + corpsB.masse) in
(*Application du TCM*)
  let VxG = (corpsA.masse * corpsA.vitesse_x + corpsB.masse *
corpsB.vitesse_x)/(corpsA.masse + corpsB.masse)
  and VyG = (corpsA.masse * corpsA.vitesse_y + corpsB.masse *
corpsB.vitesse_y)/(corpsA.masse + corpsB.masse) in

  let G = {masse = 0.; rayon = 5; coord_x = xG ; coord_y = yG; vitesse_x =
VxG; vitesse_y = VyG} in
```

```
(*Particule Fictive M*)
  let Mu = corpsA.masse * corpsB.masse / (corpsA.masse + corpsB.masse) in
  let Vx = corpsB.vitesse_x - corpsA.vitesse_x + G.vitesse_x in
  let Vy = corpsB.vitesse_y - corpsA.vitesse_y + G.vitesse_y in
  let xM = corpsB.coord_x - corpsA.coord_x + G.coord_x in
  let yM = corpsB.coord_y - corpsA.coord_y + G.coord_y in
  let M = {masse = corpsA.masse*corpsB.masse; rayon = 5; coord_x = xM;
coord_y = yM; vitesse_x = Vx; vitesse_y = Vy} in
```

On programme ensuite l'équation différentielle du problème avec ses composantes vectorielles séparées. Cette partie est véritablement le cœur du programme.

```
(*Equation différentiel en x'' et y'' de M*)

(*En x*)
  let f''x(y,x,v) =
    -(Gr * (M.masse)*(x-G.coord_x)) / ((sqrt(((x-G.coord_x)*(x-G.coord_x) +
(y-G.coord_y)*(y-G.coord_y)))^^^^3.) * Mu ) in

(*En y*)
  let f''y(x,y,v) =
    -(Gr * (M.masse)*(y-G.coord_y)) / ((sqrt(((x-G.coord_x)*(x-G.coord_x) +
(y-G.coord_y)*(y-G.coord_y)))^^^^3.) * Mu ) in
```

La déduction des trajectoires des deux corps et de leurs vitesses instantanées est traduite par des fonctions déduites des relations démontrées précédemment, composantes par composantes.

```
(*Déduction des coordonnées corpsA et corpsB par homothétie*)
(*Corps A*)
  let fxcorpsA = function x -> - (corpsB.masse * (x-G.coord_x)) /
(corpsA.masse + corpsB.masse) in
  let fycorpsA = function y -> - (corpsB.masse * (y-G.coord_y)) /
(corpsA.masse + corpsB.masse) in
(*Corps B*)
  let fxcorpsB = function x -> (corpsA.masse * (x - G.coord_x)) /
(corpsA.masse + corpsB.masse) in
  let fycorpsB = function y -> (corpsA.masse * (y - G.coord_y)) /
(corpsA.masse + corpsB.masse) in
(*Déduction des vitesses corpsA et corpsB par homothétie*)
(*Corps A*)
  let fvxcorpsA = function x -> - (corpsB.masse * (x-G.vitesse_x)) /
(corpsA.masse + corpsB.masse) in
  let fvycorpsA = function y -> - (corpsB.masse * (y-G.vitesse_y)) /
(corpsA.masse + corpsB.masse) in
(*Corps B*)
  let fvxcorpsB = function x -> (corpsA.masse * (x - G.vitesse_x)) /
(corpsA.masse + corpsB.masse) in
  let fvycorpsB = function y -> (corpsA.masse * (y - G.vitesse_y)) /
(corpsA.masse + corpsB.masse) in
```

On introduit ensuite deux objets *options*, *state* de type respectif *options* et *state* :

Ces objets assurent une interaction entre l'utilisateur et le programme à tout moment de la simulation. En effet, si l'objet *options* interviendra dans la sauvegarde des données reçue avec l'interface clavier, l'objet *state* enregistrera à tout instant l'état de la simulation (informations sur les corps, temps de simulation, informations sur les options de simulation et les échelles de représentation…).

```
(*Déclaration de "options" et "state"*)
  let options = ref {dt = dtt ; zoom = 1.; origine_x = 0; origine_y = 0;
scale_F = 0.; scale_v = 0.;
  dispTime = false; dispState = false; dispTraj = false ; dispM = false;
dispG = false; blinker = false ;
  dispCorps = true; hypermode = false; lineto = false; elag = false; reset
= false} in
```

```
  let state = {corps1 = corpsA; corps2 = corpsB; partFict = M; centreG = G;
F = 0. } in
```

On choisit ensuite de définir quatre listes pour enregistrer les trajectoires respectives des corps 1 et 2 ainsi que les trajectoires du centre de masse G et de la particule fictive M.
La présence de liste dans un programme itératif se justifie par leur souplesse, et leur flexibilité d'utilisation (dimension extensible et « non limitée »).
On ajoute un compteur $count$ initialisé à 0 qui servira dans l'enregistrement de coordonnées successives (trajectoires) tous les x itérations, où x est fonction de la vitesse de simulation.

```
(*Trajectoire int*int list.*)
  let la = ref [] and lb = ref [] and lm = ref [] and lg = ref [] in
  (*Compteur*)
  let count = ref 0 in
```

Pour une meilleure lisibilité du programme, on introduit les variables vitesses et coordonnées initiales :

```
(*Définitions obsolètes - Initialisation*)
  let yo = M.coord_y and vyo = M.vitesse_y
  and xo = M.coord_x and vxo = M.vitesse_x in
(*End_def*)
```

On souhaite afficher au cours de la simulation les vecteurs forces et vitesses. Un problème se pose quant à leurs représentations. En effet, les systèmes étudiés peuvent être très différents (par leurs masses, distances relatives…). Ainsi il n'est pas possible de prévoir une échelle statique pour réduire l'ordre de grandeur de ces vecteurs. On va donc utiliser la fonction low_number définie dans les fonctions auxiliaires. On décide de limiter la taille des vecteurs vitesses à 300 pixels. La taille limite des vecteurs forces devra être inférieure à la moitié de la distance entre les deux corps dans leurs états initiaux. Cela ne pourra donc empêcher des cas de chevauchements de ces vecteurs ou leurs représentations partielles dans le cas où la distance entre les deux corps viendrait à diminuer fortement. La fonction low_number renvoie donc une échelle exploitable qui sera enregistrée dans l'objet $options$ ($options.scale\_v$ pour les vecteurs vitesses et $options.scale\_F$ pour le vecteur force).

```
(*Déclaration et initialisation du vecteur force composantes par
composantes*)
  let fx = ref 0. and fy = ref 0. in
(*Calcule de F 1 -> 2 à t0*)
  fx := (abs_float (f''x(yo,xo,vxo)) * Mu);
  fy := (abs_float (f''y(xo,yo,vyo)) * Mu);
(*Evaluation d'une échelle appropriée*)
  (*Force*)
  !options.scale_F <- (low_number (max !fx !fy) ((sqrt ( ((corpsB.coord_x -
corpsA.coord_x)^^^^2.) + ((corpsB.coord_y - corpsA.coord_y)^^^^2.) ))/2.));
  (*Vitesse*)
  !options.scale_v <- (low_number (max vxo vyo) (300.));
```

On définie huit nouvelles variables (x, y, x', y'; xx, yy, xx', yy') pour repérer la position de la particule fictive M respectivement aux temps t_n et t_{n+1}. Il est nécessaire de faire une double déclaration des variables. En effet, l'équation différentielle est fonction de x_n et y_n. Les données seraient alors erronées dans le cas ou on calculerait y_{n+1} en fonction de x_{n+1} (au lieu de x_n).

```
(*Initialisation des coordonnées et de la vitesse de M*)
  let y = ref yo and y' = ref vyo and x = ref xo  and x' = ref vxo  and t =
ref 0. in
  let yy = ref yo and yy' = ref vyo and xx = ref xo  and xx' = ref vxo  in
```

La simulation sera démarrée par l'appui sur une touche (quelconque). La simulation sera donc initialement mise en pause à l'aide de la fonction $wait()$.

```
  (*Pause avant de commencer*)
  wait();
```

Le programme comportera une boucle « infinie » de type $while\ true$. L'utilisation d'une boucle infinie est justifiée par l'ajout de la gestion des exceptions. Ainsi pour sortir de cette boucle, il suffira de lever l'exception $Exit$ (Prédéfinie en CAML). Cette exception sera rattrapée par une structure $try \ldots with$.
La résolution de l'équation différentielle par la méthode de Runge Kutta (à l'aide de la fonction $Runge\_Kutta\_4\_ordre\_2$) est effectuée en début de boucle. Les informations seront ensuite traitées.

```
  try
    while true do
      (*Calcule de x*)
      (xx, xx') ~:= (Runge_Kutta_4_ordre_2 f''x !x !x' !y !options);
      (*Calcule de y*)
      (yy, yy') ~:= (Runge_Kutta_4_ordre_2 f''y !y !y' !x !options);
      (*Vitesse et coordonnée de M*)
      (x, x')   ~:= (!xx, !xx');
      (y, y')   ~:= (!yy, !yy');
```

Le traitement des données (calculs relatifs aux corps 1 et 2, à la particule fictive et au centre de masse) ainsi que leurs représentations graphiques se font selon le schéma suivant :

- Effacement des corps (réels et fictifs) en fonction de leurs anciennes caractéristiques.
- Mise à jour des caractéristiques des corps.
- Calcul de $F_{1/2}$.
- Mise à jour de l'état de la simulation (objet $state$).
- Affichage des corps en fonction de leurs nouvelles caractéristiques.
- Enregistrement des trajectoires.
- Affichage des trajectoires.

Pour répondre à toutes les options, cette structure est légèrement fragmentée. Toute opération non indispensable, qui peut être susceptible de ralentir la simulation, est introduite à l'intérieur d'une structure conditionnelle qui ne sera pas traitée en « Hypermode ». Ceci explique, par exemple, la fragmentation de la mise à jour des coordonnées avec la mise à jour des vitesses des corps.

```
    (*Traitement des données*)
      (*Effacement des corps*)
        if not(!options.hypermode)
          then
          begin
            (*"Réels"*)
            effaceCorp corpsB (!fx)    (!fy) !options;
            effaceCorp corpsA (- !fx) (- !fy) !options;
            (*"Fictifs"*)
            if !options.dispM then effaceCorp M (0.) (0.) !options;
            if !options.dispG then effaceCorp G (0.) (0.) !options;
          end;
```

```
        (*Mise à jour des Corps (Fictifs et simulés)*)
        (*Centre de Masse G - coordonnées*)
          G.coord_x <- (VxG * !t + xG);
          G.coord_y <- (VyG * !t + yG);
        (*Corps A - coordonnées*)
          corpsA.coord_x <- (G.coord_x + fxcorpsA !x);
          corpsA.coord_y <- (G.coord_y + fycorpsA !y);
        (*Corps B - coordonnées*)
          corpsB.coord_x <- (G.coord_x + fxcorpsB !x);
          corpsB.coord_y <- (G.coord_y + fycorpsB !y);
        (*Particule fictive - coordonnées*)
          M.coord_x <- !x;
          M.coord_y <- !y;
          if not(!options.hypermode)
          then
          begin
            (*calcule de F 1/2*)
            fx := (f''x(!y,!x,!x') * Mu);
            fy := (f''y(!x,!y,!y') * Mu);
            (*Particule Fictive M Vitesse*)
            M.vitesse_x <- !x';
            M.vitesse_y <- !y';
            (*Corps A - Vitesse*)
            corpsA.vitesse_x <- (G.vitesse_x + fvxcorpsA !x');
            corpsA.vitesse_y <- (G.vitesse_y + fvycorpsA !y');
            (*Corps B - Vitesse *)
            corpsB.vitesse_x <- (G.vitesse_x + fvxcorpsB !x');
            corpsB.vitesse_y <- (G.vitesse_x + fvycorpsB !y');
            (*Actualisation de l'état de la simulation*)
            state.corps1   <- corpsA;
            state.corps2   <- corpsB;
            state.partFict <- M;
            state.centreG  <- G;
            state.F        <- (sqrt((!fx ^^^^ 2.)+(!fy ^^^^2.)));
            (*Affichage des corps*)
            (*"Réels"*)
            drawCorp corpsB bleu (!fx) (!fy) !options;
            drawCorp corpsA marron (- !fx)  (- !fy) !options;
            (*"Fictifs"*)
            if !options.dispM then drawCorp M vert (0.) (0.) !options;
            if !options.dispG then drawCorp G rouge (0.) (0.) !options;
          end;
      (*Enregistrement des trajectoires tous les x calculs*)
        if (!count mod gt1(100. * dtt /(!options.dt)) == 0) then
          begin
          (*Enregistrement*)
          la := (op_list ((corpsA.coord_x,corpsA.coord_y) :: !la)
!options);
          lb := (op_list ((corpsB.coord_x,corpsB.coord_y) :: !lb)
!options);
          lm := (op_list ((M.coord_x,M.coord_y) :: !lm) !options);
          lg := (op_list ((G.coord_x,G.coord_y) :: !lg) !options);
          !options.reset <- false; !options.elag <- false;
          (*Affichage des trajectoires*)
          if (!options.dispTraj && not(!options.hypermode)) then
            begin
            plotlist !la marron !options;
            plotlist !lb bleu !options;
            if !options.dispM then (plotlist !lm vert !options;);
            if !options.dispG then (plotlist !lg rouge !options;);
            end
```

```
        end
      else ();
      incr count;
```

N.B :

- Il y a une dépendance de la vitesse de simulation dans l'enregistrement des trajectoires. En effet, selon les variations du pas d'itération, on aurait des trajectoires représentées par des points de façon plus ou moins espacés. Cette dépendance tend donc à uniformiser les trajectoires dans la mesure du possible (si le pas d'itération devient trop grand, les trajectoires seront enregistrées à chaque tour de boucle ; $count$ est le compteur de tour de boude : « $incr\ count \Leftrightarrow count := count + 1$ »).
- op_list est la fonction qui permet de réduire la longueur des listes ou de les remettre à zéro dans le cas ou leur taille deviendrait une gêne pour la vitesse de la simulation.

Il est nécessaire de vérifier à la fin de la boude si les corps ne sont pas entrés en collision. La simulation pourrait devenir « fausse » dans le cas contraire. Il y a collision dans le cas ou la distance entre les deux corps est inférieure à la somme de leurs rayons. Si c'est le cas, on lève alors l'exception $collision$ qui mettra fin à la simulation.

```
  (*Vérifications des collisions*)
      if ((sqrt (((corpsB.coord_x - corpsA.coord_x)^^^^2.) +
((corpsB.coord_y - corpsA.coord_y)^^^^2.)))  < float (corpsA.rayon ++
corpsB.rayon))
        then raise collision;
```

Arrivés en fin de boucle, on augmente le temps de la simulation en lui ajoutant le pas d'itération puis on l'affiche selon les options de l'utilisateur.

```
      (*Temps*)
        t  := !t + !options.dt;
      (*Affichage du temps*)
        if  (!options.dispTime  &&  not(!options.hypermode))  then  draw_time
!t;
        if  (!options.dispState&&  not(!options.hypermode))  then  draw_state
(vectString_of_state state !options);
```

Il ne reste plus qu'à gérer les interactions utilisateur qui se font à l'aide de la fonction $waitif$ décrite ci-dessus.

```
(*Interaction utilisateur*)
        options := waitif !options state;
```

On notera l'existence de l'appel à la fonction $pause$ dans le cas ou l'option visant à réduire le clignotement de la fenêtre graphique est activée. Cette fonction a uniquement pour but de réduire sensiblement le programme pour permettre un rendu graphique plus fluide.

```
(*Anti-blink - Mode visant à atténuer le clignotement de l'écran*)
      if (!options.blinker && not(!options.hypermode)) then pause 1;
  done;
```

Les exceptions, éventuellement, levées sont rattrapées par la structure $try \ldots with$ à la fin du programme.

```
with
  |Exit -> print_string "Fin de la simulation."
  |collision -> bang(); print_string "BANG";;
```

B° Difficultés rencontrées

1) Difficultés liées au problème physique

Une bonne compréhension du problème physique est indispensable pour le résoudre. En effet, l'algorithme de la partie principale, une fois les outils de résolution numérique et les fonctions auxiliaires mis en place, découle quasi-directement de l'analyse du problème fait en première partie.

Des premières versions qui semblaient concluantes se sont révélées erronées lorsque :

- Le centre de masse ne se trouvait pas à l'origine du repère (œ qui implique une erreur dans l'implémentation de l'algorithme d'une des formules).
- Le centre de masse était en mouvement (il faut, en effet, prendre en compte les variations du centre de masse qui interviennent dans la distance de la particule fictive).

Un autre problème incontournable (par sa place centrale dans le programme) fut celui de la traduction de l'équation différentielle vectorielle, en deux équations (composantes par composantes) dans le repère de la fenêtre graphique (repère cartésien). Une équation différentielle erronée décourageait d'avance toute solution. C'est donc une partie incontournable du programme.

2) Difficultés liées à la résolution d'équation différentielle d'ordre 2

Si le passage de la méthode d'Euler pour résoudre une EDO d'ordre 1 à la résolution d'une EDO d'ordre 2 se fait sans difficulté, ce fut plus compliqué pour le passage de la méthode de Runge Kutta 4 sur une EDO d'ordre 1 à une EDO d'ordre 2.

Ceci fut un des facteurs limitant, (en plus de la complexité croissante), à l'adaptation d'une des autres méthodes de Runge Kutta d'ordres supérieurs, comme celles abordées dans la partie sur les méthodes de résolutions numériques.

3) Difficultés de programmations

a) Listes et programmation récursive

La programmation récursive offre souvent un confort visuel non négligeable, sans toujours ralentir le programme. Ainsi dans les cas ou la complexité d'un algorithme récursif est du même ordre de grandeur que celle d'un algorithme itératif, une méthode récursive sera souvent privilégiée. Des objets comme les listes se traitent naturellement par des procédés récursifs.

Dans notre cas, on utilise rarement ce type de procédés, même sur les listes. Une première étude nous a conduit à penser que la taille maximum des listes était limitée. Nous en sommes arrivés à cette conclusion à l'aide du programme suivant :

```
let rec taille n =
match n with
|0 -> []
|_ -> (n)::(taille (n-1));;

taille 104796;;
taille 104797;;
```

qui retourne le résultat suivant :

```
#taille 104796;;
- : int list =
 [104796; 104795; 104794; 104793; 104792; 104791; 104790; 104789; ...]
#taille 104797;;
Uncaught exception: Out_of_memory
```

Cependant, en concaténant deux listes de la taille « maximum », on obtient une nouvelle liste de deux fois cette taille :

```
list_length( taille 104796 @ taille 104796 ) ;;
- : int = 209592
```

On en déduit (après un grand nombre de concaténation de liste de grande taille) que la longueur maximum d'une liste n'est à priori pas limitée (si ce n'est que par la mémoire de l'ordinateur).

On peut donc conclure que le nombre d'itérations maximum, dans une boucle récursive, est de $104\,796$. Toute fonction récursive susceptible de dépasser ce nombre devra être capable de rattraper l'exception levée pour ne pas faire interrompre la simulation.

Cette limite est elle atteignable dans notre programme ?

Nous avons décidé de vérifier si l'utilisation de structure $try \ldots with$ était réellement nécessaire dans notre cas. On estime donc dans un premier temps le nombre moyen de calculs par seconde effectué par une machine classique de performance modérée. Le nombre moyen est de 105.8 calculs par seconde, (cette moyenne a été calculée sur trois séries de 5000 calculs effectués en « **hypermode** » : après $5\,000$ calculs, $35\,000$ calculs et après $50\,000$ calculs), en enregistrant à chaque tour les trajectoires, ce qui n'est pas le cas en général. Dans le pire des cas, il faudrait, donc, attendre environ $16\,minutes\,et\,30\,secondes$ pour que l'exception Out_of_memory soit levée. Un tel temps de simulation n'est pas raisonnable compte tenu des erreurs accumulées. Nous avons donc décidé de ne pas rattraper cette exception dans le cas ou elle devrait être levée. Cependant, il nous est possible à tout moment de vider les listes qui contiennent les trajectoires à l'aide de la touche x.

b) Problèmes d'ordre graphique

La représentation graphique du problème en temps réel a posé quelques problèmes. En effet, CAML n'est pas un langage adapté pour créer des animations.

Après avoir résolu la difficulté de tracer les trajectoires, de manière continue, sans qu'elles soient effacées par la traversé d'un corps, un autre problème persistait.

La vitesse des affichages et des nettoyages successifs des corps cause un effet de « clignotement » de la fenêtre graphique. La seule solution probante fut de ralentir considérablement la simulation, pour avoir un rendu visuel convenable. On devait donc faire un compromis entre vitesse et rendu graphique. Finalement, ce choix est soumis à l'utilisateur qui peut activer ou non le mode « anti-blink », mode qui réduit considérablement ce phénomène de clignotement. Il est toujours possible, d'accélérer la simulation avec ce mode activé. La qualité graphique reste nettement améliorée, cependant, les erreurs dues à l'approximation s'en trouvent agrandies.

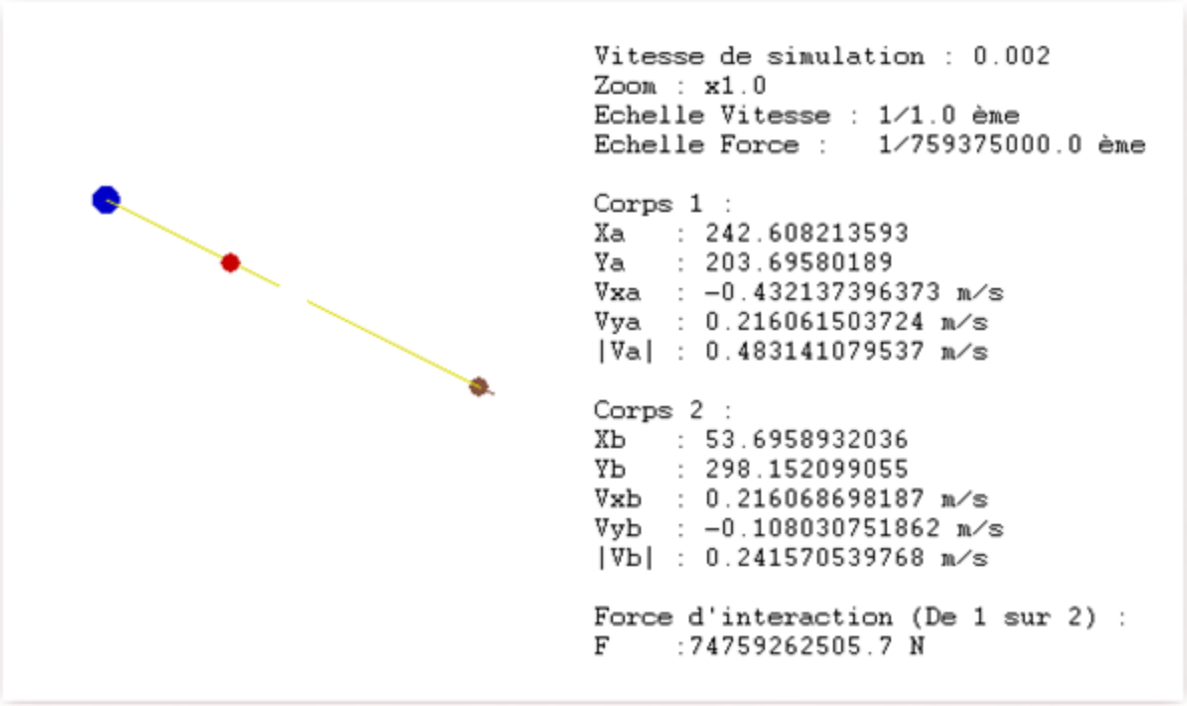
Le problème de la « 3 dimension » ne s'est, dans un premier temps, pas posé grâce à la planéité du mouvement. Cependant, nous avons voulu représenter les corps par des sphères, et non simplement par des cercles. Ce problème, loin d'être extrêmement compliqué, s'est révélé laborieux. De plus, ce mode de représentation ralentissait encore considérablement le programme. Nous avons donc décidé, à défaut de créer une nouvelle option, de ne pas représenter les corps en 3D.

C° Exemples classiques de simulation

Nous allons tester notre programme sur des cas de difficultés croissantes. Nous commencerons par un cas trivial, celui de deux masses en interaction sans vitesse initiale, pour arriver progressivement à un cas ou le centre de masse est en mouvement. Cela correspond chronologiquement aux différents tests effectués sur notre programme, au cours de ses versions successives. On représentera selon les cas : en $rouge$ le centre de masse G et en vert la particule fictive M du système. Les trajectoires seront affichées dans la couleur des corps. Les vecteurs forces sont représentés en vert « caca d'oie » et les vecteurs vitesses en rouge. Les échelles relatives à la représentation des vecteurs et des corps ne seront pas toujours précisées. Il s'agit surtout de vérifier la conformité des trajectoires sur des cas connus.

1) Deux masses sans vitesse initiale en interaction gravitationnelle

```
(*Exemple 1*)
(*Deux masse sans vitesse initiale en interaction gravitationnelle*)

let A = {masse = 5e12; rayon = 5; coord_x = 250. ; coord_y = 200.;
vitesse_x = 0.; vitesse_y = 0.};;
let B = {masse = 1e13; rayon = 7; coord_x = 50.; coord_y = 300.; vitesse_x
= 0.; vitesse_y = 0.};;

solve A B 0.001;;
```

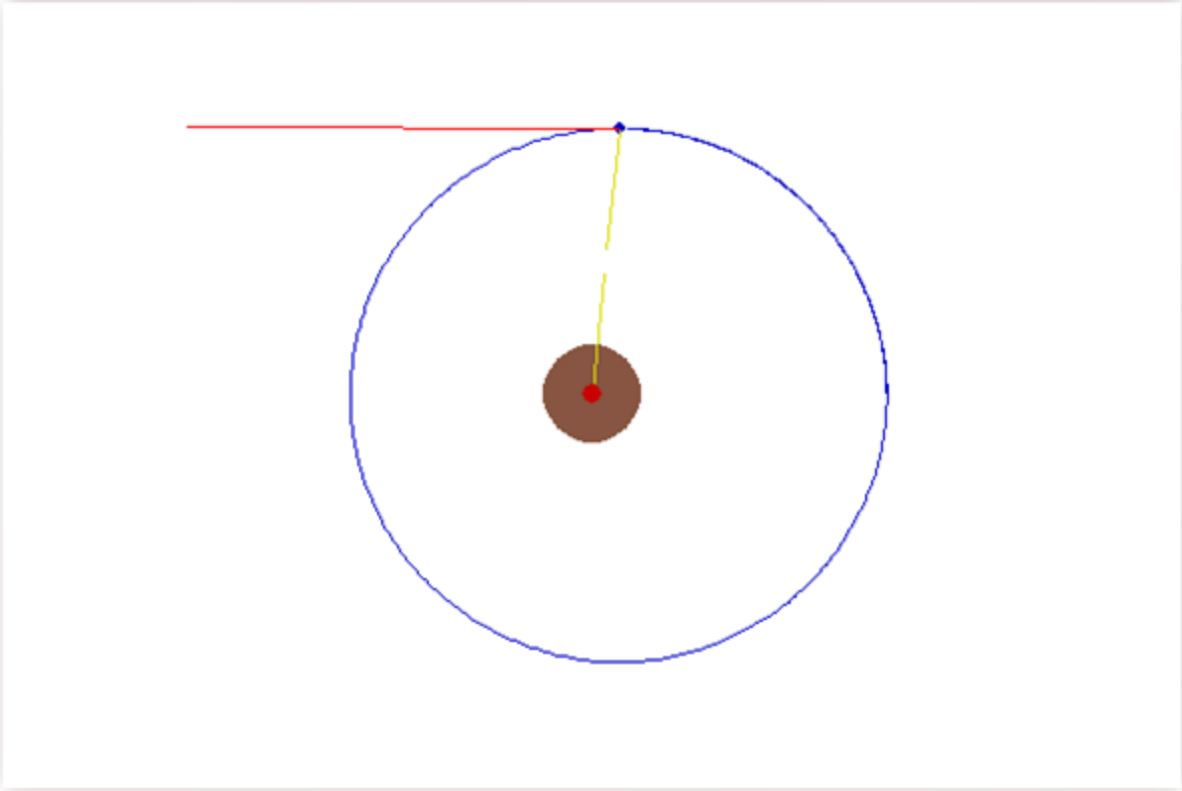


Les vecteurs vitesses sont confondus avec les vecteurs forces (Ils n'ont pas mêmes normes.)

Les deux corps s'attirent en mouvement rectiligne accéléré, comme dans le cas réel, jusqu'à entrer en collision (les informations fournies correspondent au cas réel, avec une certaine approximation.) On note, cependant, l'excès de chiffres significatifs que nous avons malgré tout gardés dans le cadre de notre étude. Ce programme fait une simulation de cas réels (ou fictifs). Compte-tenu de l'approximation due à la résolution numérique, on ne peut savoir exactement le nombre de chiffres significatifs.

2) Cas d'une masse très négligeable devant la seconde

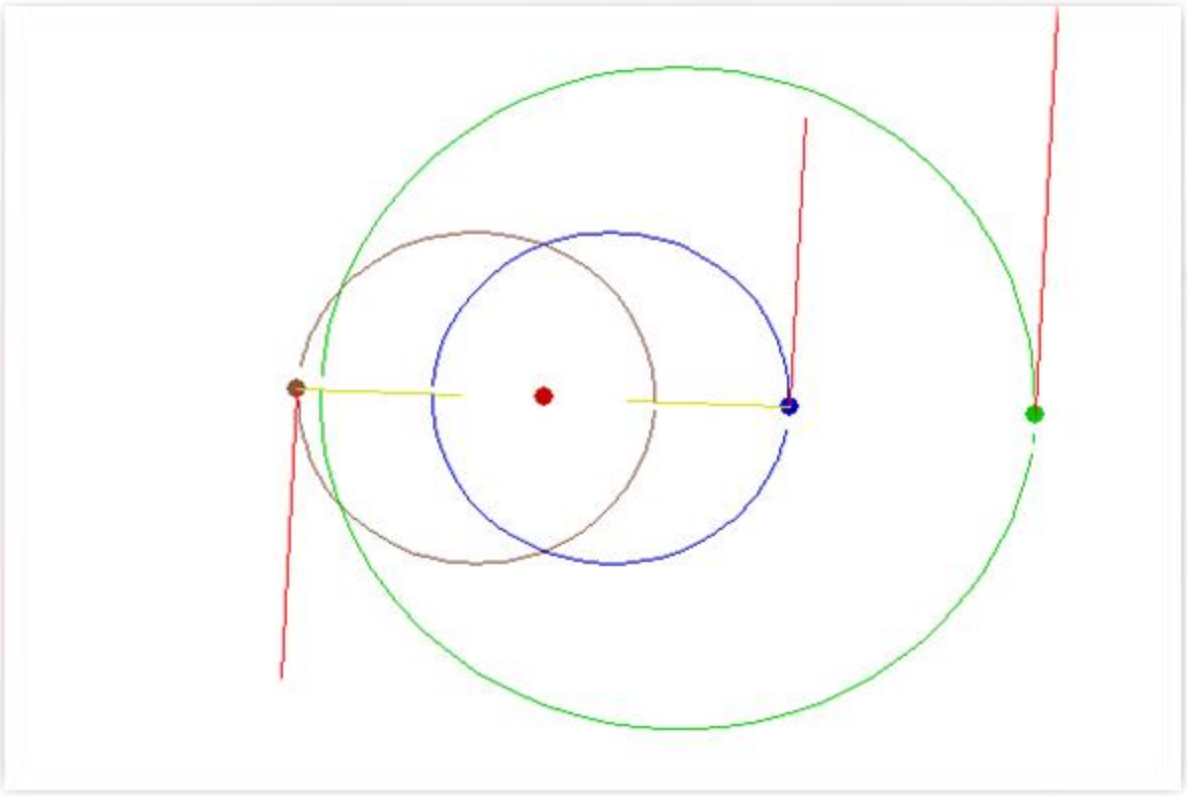
```
(*Exemple 2*)
(*Cas d'une masse négligeable devant la seconde*)

let A = {masse = 1e17; rayon = 25; coord_x = 300.; coord_y = 200.;
vitesse_x = 0.; vitesse_y = 0.};;
let B = {masse = 300.; rayon = 3;  coord_x = 450.; coord_y = 200.;
vitesse_x = 0.; vitesse_y = 200.};;

solve A B 0.0001;;
```



Le centre de masse est confondu avec le $corps A$, (la particule fictive est donc confondue avec le $corps B$.) On est, donc, dans le cas d'un problème à force centrale.

3) Deux masses égales en interaction

```
(*Exemple 3*)
(*Deux masses égales en interaction*)

let A = {masse = 3e18; rayon = 5; coord_x = 150.; coord_y = 200.; vitesse_x
= 0.;  vitesse_y = -500. };;
let B = {masse = 3e18; rayon = 5; coord_x = 400.; coord_y = 200.; vitesse_x
= 0.; vitesse_y = 500.};;

solve A B 0.0001;;
```

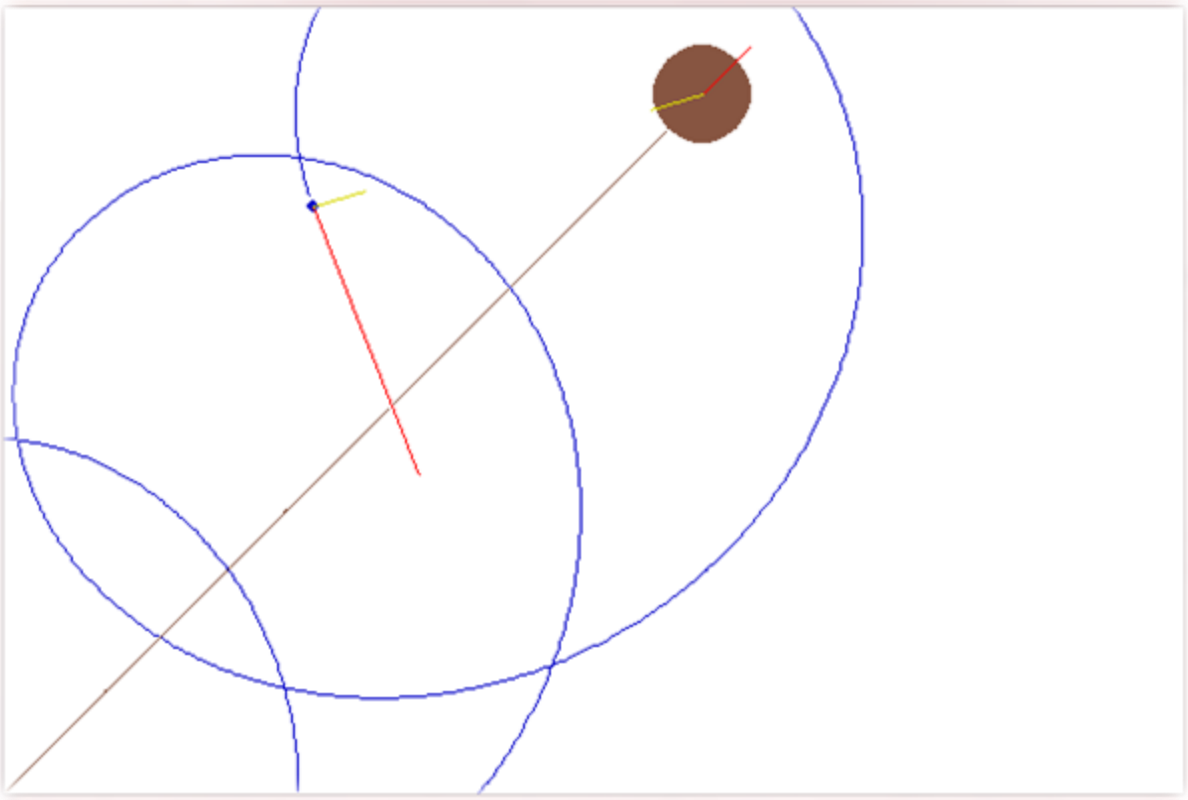


On simule l'interaction de deux masses égales en interaction. Les trajectoires obtenues sont elliptiques. En modifiant un peu les conditions initiales, on obtient deux masses égales sur une orbite circulaire (Voir exemple 3 bis – Exemple.ml).

4) Cas ou le centre de masse est en mouvement

On se replace dans les conditions de l'exemple b, On donne une vitesse initiale au $corpsA$ (qui est équivalent au centre de masse G). Le centre de masse sera, donc, en mouvement rectiligne uniforme.

```
(*Exemple 4*)
(*Centre de masse en Mouvement rectiligne uniforme*)

let A = {masse = 1e17; rayon = 25; coord_x = 0. ; coord_y = 0.; vitesse_x =
25.; vitesse_y = 25.};;
let B = {masse = 300.; rayon = 3; coord_x = 150.; coord_y = 0.; vitesse_x =
0.; vitesse_y = 250.};;

solve A B 0.0001;;
```



D° Limites et extensions du programme

1) Limites

a) Erreurs accumulées

Les erreurs accumulées lors de la résolution de l'équation différentielle rendent difficile les simulations sur de longues périodes de temps. De plus, lorsque deux corps sont très proches et qu'ils sont soumis à des forces importantes, il devient indispensable de réduire grandement le pas d'itération. Dans le cas contraire, la simulation peut être faussée et peut afficher une trajectoire hyperbolique alors qu'avec un pas adapté, la trajectoire est elliptique.

b) Gestion des collisions

Le programme s'arrête instantanément en cas de collision, mais que se passe-t-il dans le cas réel ? Nous allons voir pourquoi, nous avons décidé de ne pas aborder ce problème.

Les collisions entre deux corps peuvent se faire de deux sortes :

- Sous la forme de collisions élastiques (ou quasi-élastiques) : elles se caractérisent par l'absence de perte d'énergie. C'est le cas pour deux solides indéformables (sphère dure). Il y a alors conservation de l'énergie cinétique et conservation de la quantité de mouvement. On obtient un système d'équation tout à fait programmable. On trouve une illustration de ce phénomène dans le pendule de Newton.

- Sous la forme de collisions inélastiques : Il y a conversion de l'énergie cinétique en énergie interne qui sera utilisée pour déformer les corps. C'est le cas pour des solides déformables. Le problème est alors bien plus compliqué et ne pourrait être envisagé dans ce TIPE.

La simulation de collisions élastiques serait elle raisonnable ?

Dans le cadre de la gravitation, il est impossible de considérer les corps comme des sphères infiniment dures. La modélisation des collisions par des chocs élastiques est donc inappropriée. C'est pour cela que nous décidons d'arrêter le programme en cas de collision.

2) Extensions possibles

Système de deux particules chargées isolées :

Le système est analogue à deux masses en interaction. Peu de changements sont à effectuer. On a :

$$\frac{d^2(\overrightarrow{GM})}{dt^2} = \frac{m_A + m_B}{m_A m_B} \cdot \overrightarrow{F_{A/B}}$$

Avec $\overrightarrow{F_{A/B}}$ la force de Coulomb:

$$\overrightarrow{F_{A/B}} = -\frac{1}{4\pi\varepsilon_o} \cdot \frac{q_A q_B}{AB^2} \cdot \overrightarrow{e_{A\to B}}$$

Dans ce cas : Il suffirait, pour adapter notre programme, de remplacer la constante de gravitation Gr par $\frac{1}{4\pi\varepsilon_o} \approx 9 \times 10^9 S.I$ avec ε_o la permittivité électrique du vide. Les masses respectives de A et B deviendraient alors les charges.

Modèle de la molécule diatomique :

On peut modéliser une molécule diatomique par un système de deux points matériels relié par un ressort de longueur à vide $l_0 = 0$.

On obtient une équation différentielle, par un raisonnement analogue à la réduction canonique de notre problème, de la forme :

$$\frac{d^2(\overrightarrow{GM})}{dt^2} = \frac{m_A + m_B}{m_A m_B} \cdot \overrightarrow{F_{A/B}}$$

Avec :

$$\overrightarrow{F_{A/B}} = -k\,(l - l_0) \cdot \overrightarrow{e_{A\to B}}$$

IV Résultats expérimentaux

A° Simulation Terre-Lune

1) Données

On se propose de vérifier la période de révolution de La Lune autour de La Terre. On considère la Terre et la Lune comme étant isolées dans le référentiel d'étude. (Réduction à un problème à deux corps).

	Terre	Lune
Masse	$5{,}9736 \times 10^{24} Kg$	$7{,}3477 \times 10^{22} Kg$
Rayon	$6{,}3781 \times 10^{3} Km$	$1{,}7374 \times 10^{3} Km$
Vitesse orbitale moyenne	$\times$	$1{,}022\ Km.s^{-1}$
Période de révolution	$\times$	$\mathbf{2{,}361 \times 10^{6} s}$
Distance Terre-Lune	$384\ 400\ km$	

2) Simulation informatique

Les unités de la simulation sont celles du Système International (il sera nécessaire de dézoomer un bon nombre de fois pour avoir un rendu correct).

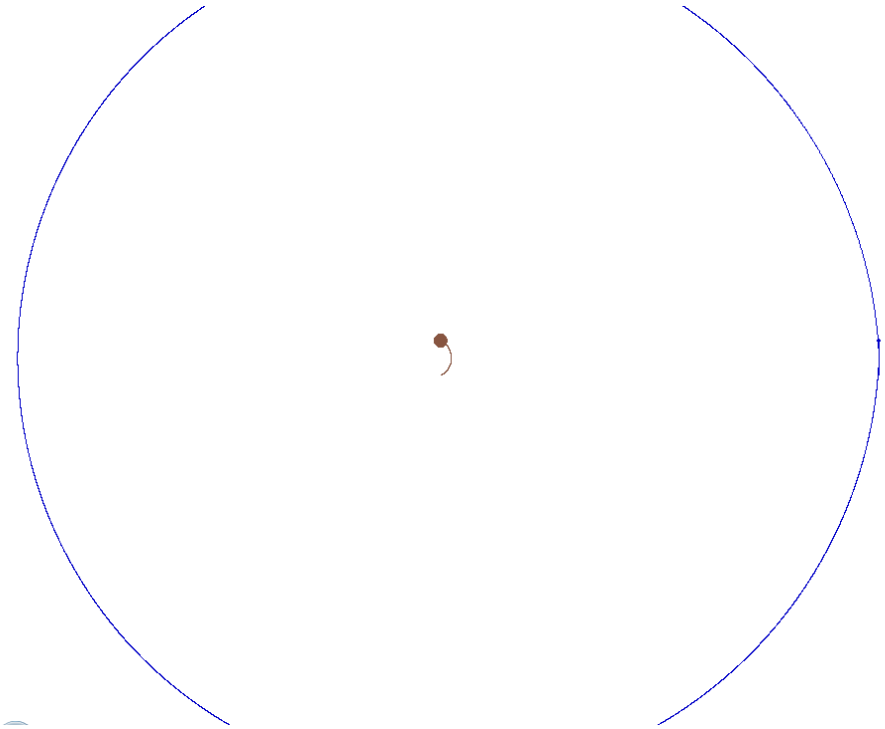
```
let Terre = {masse = 5.9736e24; rayon = 6378100; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Lune  = {masse = 7.3477e22; rayon = 1737400; coord_x = 384.4e6;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 1.022e3};;

open_graph "";;
solve Terre Lune 1.;;
```

Pour une vitesse respectable de simulation, On fixe le pas d'itération à 1. Même si cela peut paraître important, il n'en est rien à l'échelle de la simulation.

Remarque : Il est conseillé de passer en « hypermode » pour réduire le temps de la simulation.

NB : On obtient encore de très bons résultats avec un pas fixé à 10.

3° Exploitation des résultats



```
Temps  : 2341827.01172 s.
Vitesse de simulation : 0.015625
Zoom : x1.15187816648e-006
Echelle Vitesse : 1/10.0 ème
Echelle Force :   1/1e+018 ème

Corps 1 :
Xa   : -14.7877292465
Ya   : 29581140.2193
Vxa  : 3.40957484911e-005 m/s
Vya  : 1.97064262455e-005 m/s
|Va| : 3.93810017716e-005 m/s

Corps 2 :
Xb   : 384401202.226
Yb   : 29582192.6032
Vxb  : -0.00277194718328 m/s
Vyb  : 1009.58025041 m/s
|Vb| : 1009.58025041 m/s

Force d'interaction (De 1 sur 2) :
F    :1.98127098924e+020 N
```

On peut vérifier qu'on a bien effectué une rotation complète autour de la Terre ($y_A \approx y_B$).

Le résultat obtenu pour le temps de rotation de la Lune autour de la Terre par simulation est de $2{,}341 \times 10^6$ secondes. Ce résultat est très proche du résultat théorique donné : $2{,}361 \times 10^6 s$.

On se propose de calculer le taux d'erreur du résultat donné par la simulation :

$$\varepsilon = \left| \frac{valeur_{théorique} - valeur_{expérimentale}}{valeur_{théorique}} \right|$$

Application numérique :

$$\varepsilon = 8.471 \times 10^{-3}$$

$$\boldsymbol{\varepsilon = 0{,}85\ \%}$$

On constate donc qu'on a un très bon résultat obtenu par simulation. Ce résultat nous conforte dans le choix de la méthode de Runge Kutta 4 pour résoudre l'équation différentielle du problème.

B° Mise en orbite d'un satellite terrestre et vitesse cosmique

On cherche à déterminer la première et deuxième vitesse cosmique pour un satellite terrestre à l'aide de notre programme.

1) Première vitesse cosmique

La première vitesse cosmique v_o correspond à la vitesse minimale pour que le satellite puisse faire le tour de l'astre. Cela correspond à une trajectoire circulaire de rayon R_T autour de la Terre (où R_T représente le rayon de la Terre). On détermine cette vitesse à l'aide du programme, en augmentant progressivement la vitesse de lancé du satellite. On sait que cette vitesse ne dépend pas de la masse du satellite. Pour avoir un temps de simulation suffisamment cours, on fixera la masse à $1 \times 10^3 kg$, (arbitrairement), ordre de grandeur de la masse d'un satellite artificiel.

On proposera un encadrement de la vitesse v_o.

Simulations :

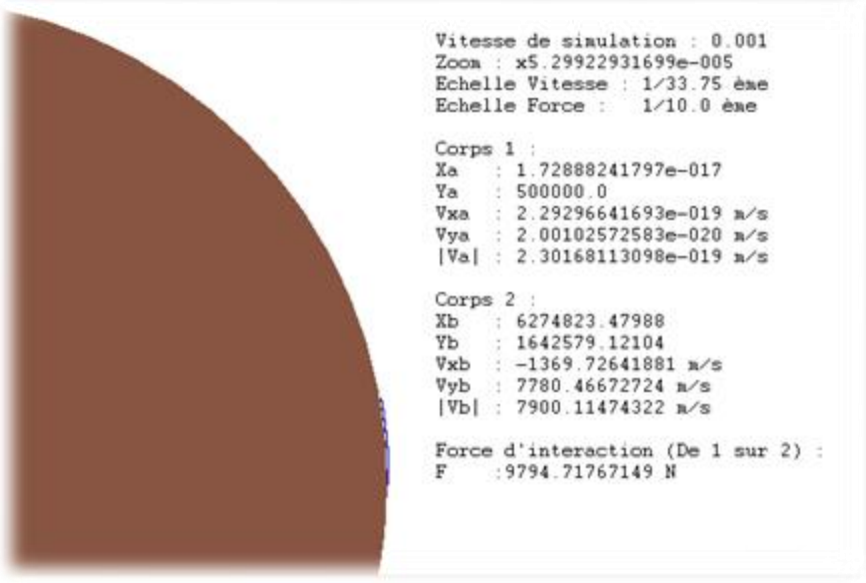
```
(*Première vitesse cosmique*)
(*1 - Retombé sur Terre du satellite*)

let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat  = {masse = 1e3; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 7.900e3};;

solve Terre Sat 0.001;;

(*2 - Mise en orbite correcte*)

let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat  = {masse = 1e3; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 7.950e3};;

solve Terre Sat 0.001;;
```

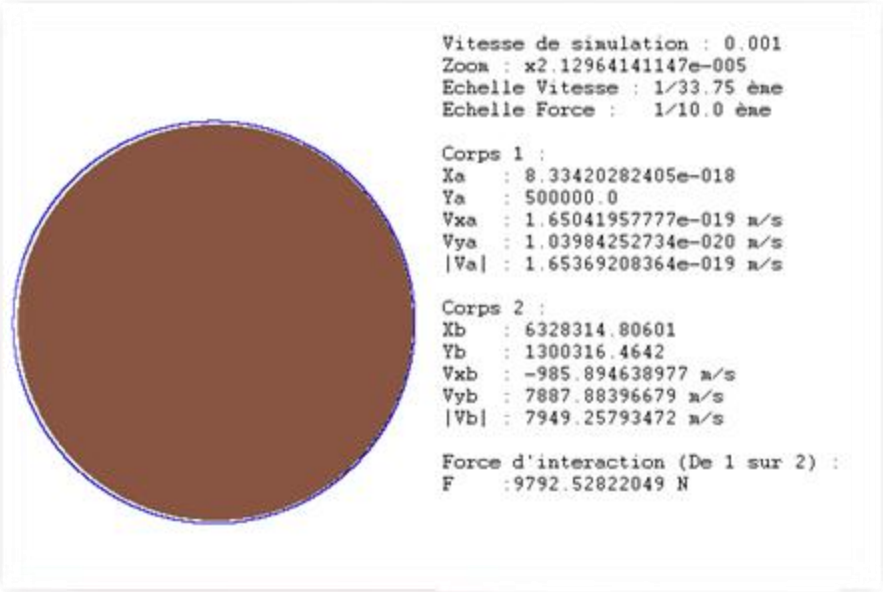
On fixe dans les deux cas le pas d'itération à $dt = 0{,}001$ ce qui devrait nous donner une assez bonne approximation. On place le satellite à une altitude initiale de $100\ mètres$.

Dans la première simulation, l'exception collision est levée : le satellite retombe sur Terre. Il ne sera pas mis en orbite.

Nous prendrons donc cette vitesse pour minorer la première vitesse cosmique :

$$v_1 = 7{,}90 \cdot 10^3 m.s^{-1} < v_o$$





Dans la deuxième simulation, le satellite effectue un tour complet autour de la Terre sans entrer en collision avec celle-ci. On considère donc qu'il a été correctement mis en orbite.

Nous prendrons cette vitesse pour majorer la première vitesse cosmique :

$$v_o < v_2 = 7{,}95 \cdot 10^3 m.s^{-1}$$

On a alors :

$$7{,}90 \cdot 10^3 m.s^{-1} < v_o < 7{,}95 \cdot 10^3 m.s^{-1}$$

La Première vitesse cosmique est donnée théoriquement par la relation suivante :

$$v_0 = \sqrt{\frac{Gr \cdot m_T}{R_T}}$$

Avec :

- Gr la constante gravitationnelle ($Gr = 6.67 \cdot 10^{-11}\ SI$)
- m_T la masse de la Terre ($m_T = 6{,}0 \cdot 10^{24}\ kg$)
- R_T le rayon de la Terre ($R_T = 6370 \cdot 10^3 m$)

Application numérique :

$$v_o = 7{,}92 \cdot 10^3 m.s^{-1}$$

On a donc réussi à avoir une approximation correcte à 2 chiffres significatifs.

2) Deuxième vitesse cosmique

La seconde vitesse cosmique (où vitesse de libération) v_{lib}correspond à la vitesse minimale pour que le satellite échappe à l'attraction de l'astre et s'éloigne infiniment de ce dernier. Cela correspond à une trajectoire parabolique. On détermine cette vitesse à l'aide du programme, en diminuant progressivement la vitesse de lancé du satellite. On sait que cette vitesse ne dépend pas de la masse du satellite. On fixera la masse à $1 \times 10^3 kg$, comme dans le cas précédent. On proposera un encadrement de la vitesse v_{lib}.

Une difficulté supplémentaire s'ajoute au problème dans le cas ou l'excentricité de la conique e tend vers 1^-, cas d'une ellipse très allongée. On considèrera la trajectoire comme parabolique si une de ses branches peut être approximée convenablement par une asymptote. Dans le cas contraire, on

considèrera que la trajectoire est elliptique. La durée de simulation sera assez importante compte-tenu du pas d'itération et de la longueur de l'intervalle d'étude. On souhaite avoir dans le cas d'une ellipse, une trajectoire quasi-complète du satellite. Il est donc conseillé de passer en « hypermode ». Il pourra aussi être nécessaire d'élaguer les $listes$ d'enregistrements des trajectoires, compte-tenu de la place en mémoire de ces dernières :

Le gestionnaire de tâches Windows nous donne une information sur la mémoire utilisée par CAML : environ 300 Mo aux trois-quarts de la simulation contre environ 5 Mo en temps normal.

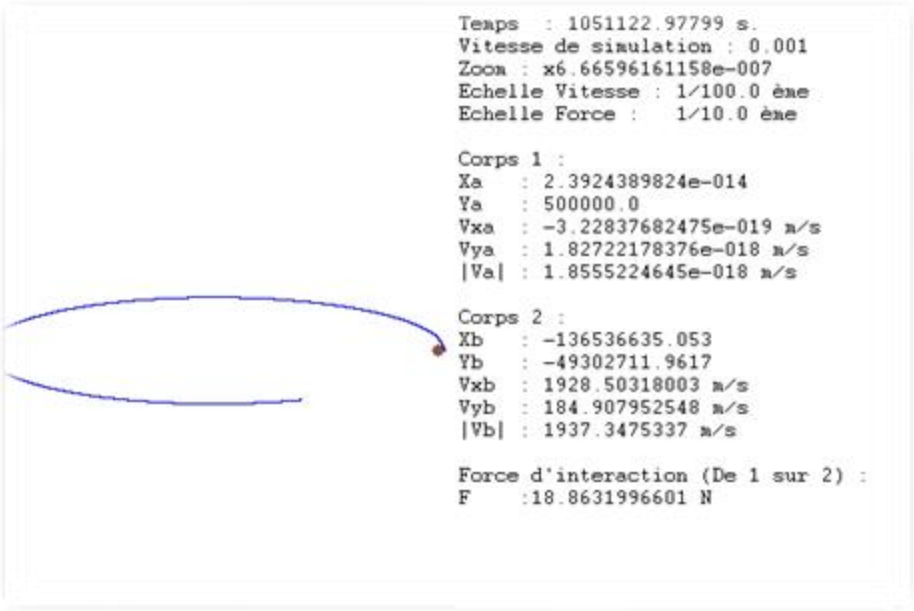
Nom de l'image	Nom d'u...	Processeur	Mémoire (jeu de travail...	Description
camltoplevel32.exe		43	323,780 K	camltoplevel32.exe

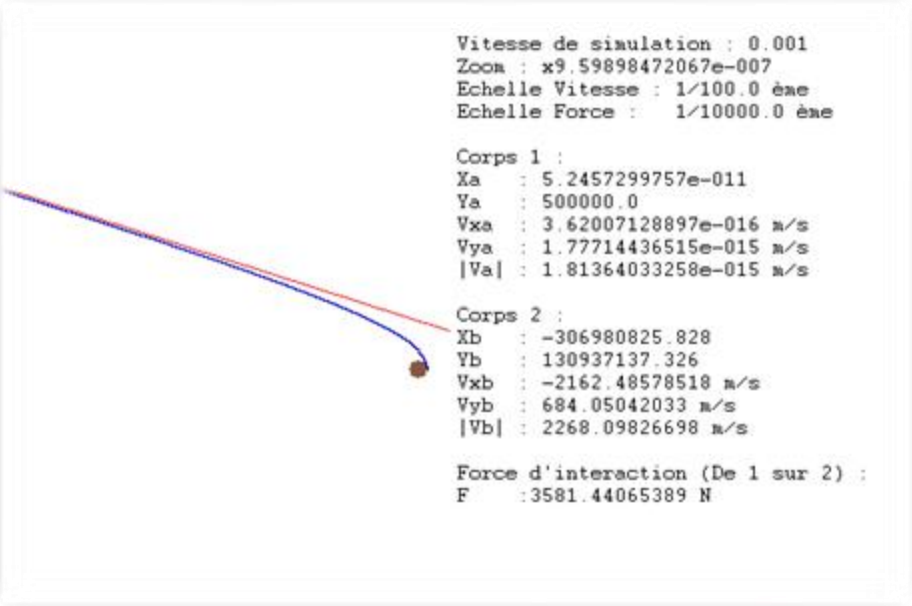
Simulations (Temps estimé de simulation : 10 $minutes$ par simulation) :

```
(*Deuxième vitesse cosmique*)

(*Echec de libération*)

let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat  = {masse = 1e3; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 11.100e3};;

solve Terre Sat 0.001;;

(*Le satellite échappe à l'attraction de la Terre*)
let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat  = {masse = 1e6; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 11.300e3};;

solve Terre Sat 0.001;;
```

La première simulation nous donne une orbite elliptique très aplatie (excentricité proche de 1^-). Nous prendrons cette vitesse pour minorer v_{lib}. Ainsi :

$v_1 = 11{,}1 \cdot 10^3 m.s^{-1} < v_{lib}$





Dans la deuxième simulation, la trajectoire est parabolique. (On représente en rouge l'asymptote). On majore ainsi la vitesse de libération v_{lib} :

$v_{lib} < v_2 = 11{,}3 \cdot 10^3 m.s^{-1}$

On a donc :

$$11{,}1 \cdot 10^3 m.s^{-1} < v_{lib} < 11{,}3 \cdot 10^3 m.s^{-1}$$

La Deuxième vitesse cosmique est donnée théoriquement par la relation suivante :

$$v_{lib} = \sqrt{\frac{2 \cdot Gr \cdot m_T}{R_T}}$$

Avec :

- Gr la constante gravitationnelle ($Gr = 6.67 \cdot 10^{-11}\ SI$)
- m_T la masse de la Terre ($m_T = 6{,}0 \cdot 10^{24}\ kg$)
- R_T le rayon de la Terre ($R_T = 6370 \cdot 10^3 m$)

Application numérique :

$$v_{lib} = 11{,}2 \cdot 10^3 m.s^{-1}$$

On a encore un résultat correct à deux chiffres significatifs.

Conclusion générale :

On a vu comment résoudre informatiquement le problème à deux corps. La simulation de ce problème découle directement des différentes relations obtenues par les théorèmes de la mécanique de Newton donc non relativiste. Les effets de la relativité seraient à prendre en compte pour des corps en interaction soumis à des vitesses très importantes (supérieures au $10^{ième}$ de la vitesse de la lumière). Le programme obtenu donne de très bons résultats et ce malgré l'utilisation d'une méthode de résolution numérique comme on a pu le voir sur des exemples concrets. De nombreuses applications sont donc possibles (calcul de la période de révolution, calcul de vitesses cosmique…).

Plus généralement, ce programme est facilement adaptable à d'autres types de problèmes à deux points matériels. On constate, cependant, qu'un tel cas est une approximation et ne permet pas d'expliquer avec précision les mouvements des corps du système solaire.

Annexes :

Annexe 1 : Interaction clavier

Annexe 2 : Commentaires du Grapheur

Sources :

Source 1 : Méthode d'euler.ml

Source 2 : Méthode de Runge Kutta.ml

Source 3 : Méthode de Runge Kutta F 5.ml

Source 4 : Représentation Graphique.ml

Source 5 : Résolution numérique.ml

Source 6 : Problème à deux corps mode continu.ml

Source 7 : Exemples.ml

Annexes

Annexe 1 :
Interaction Clavier et Description des options :

Ajustement de la fenêtre graphique :

- Fonctions de déplacement de la fenêtre graphique

- `z` : ↑ Déplacement de la fenêtre graphique le haut.
- `s` : ↓ Déplacement de la fenêtre graphique le bas.
- `q` : ← Déplacement de la fenêtre graphique vers la gauche.
- `d` : → Déplacement de la fenêtre graphique vers la droite.

- Fonctions de zoom

- `+` : Augmentation du zoom ($\times 1{,}2$)
- `-` : Diminution du zoom ($\div 1{,}2$)

- Affichage des corps et trajectoires

- `a` : **A**ffichage des trajectoires des corps.
- `c` : Relie les coordonnées des trajectoires entre elles. (Mode **c**ontinu).
- `m` : Affichage de la particule fictive $\boldsymbol{M}$.
- `g` : Affichage du centre de masse $\boldsymbol{G}$.

- Mode « Anti-blinker »

- `b` : Réduit le clignotement de l'écran par une pause dans la boucle du programme. Cela a pour principale conséquence de réduire grandement la vitesse de simulation. Le rendu visuel s'en trouve grandement amélioré.

Affichage de l'état de la simulation :

- `t` : Affichage du **t**emps de la simulation.
- `i` : Affichage des informations relatives à la simulation. Cela à pour effet de ralentir la simulation. Un clignotement des informations est possible.

Options de simulation :

- `>` : Sens normal de simulation. ($t \nearrow$).
- `<` : Sens inverse de la simulation. ($t \searrow$). Les erreurs s'accumulent. Ainsi, si on passe de $y(t_{01})$ à $y(t_1)$ puis de $y(t_1)$ à $y(t_{02})$ avec $t_{01} = t_{02}$, $y(t_{01}) \approx y(t_{02})$.
- `r` : Accélère la vitesse la simulation par 2 en agissant sur le pas d'itération. Plus la vitesse de simulation est importante, plus l'erreur commise lors de la résolution numérique est grande.
- `l` : Ralentit la vitesse de la simulation par 2 en agissant sur le pas d'itération.

Accélération de la simulation

- `x` : Supprime les trajectoires mises en mémoire. La vitesse de simulation est alors augmentée car il n'y a plus de point à tracer.
- `e` : **E**lague les listes des trajectoires. Les trajectoires perdent une coordonnée sur deux. La simulation est alors accélérée jusqu'à ce que les listes se remplissent de nouveau.

- `h` : Active / Désactive « l ' **h**ypermode ». Toutes les options susceptibles de ralentir la simulation sont désactivées. La simulation est grandement accélérée. Le programme travaille alors en circuit fermé, aucun résultat n'est affiché à l'écran pendant cette période.

Pause :

- Espace (ou toute touche ne correspondant à aucune autre fonction) : Met la simulation en Pause et affiche son état.

Arrêt de la simulation :

- `Ecp` : Met fin à la simulation et ferme la fenêtre graphique.

Annexe 2 :
Description du grapheur et commentaires sur la source :

On se propose de construire une fonction graph qui tracera la courbe représentative d'une fonction quelconque. Les fonctions seront passée en argument dans une matrice de type $float\ vect\ vect$, enregistrant les coordonnées successives de la courbe.

Nous travaillerons donc avec les bibliothèques $float$ et $graphics$ ouvertes. On redéfinira encore des opérations sur les entiers pour alléger la syntaxe.

```
#open "float";;
#open "graphics";;

(*Fonction "plot" améliorée*)

let prefix ++ a b = int_of_float(float_of_int(a) + float_of_int(b)) ;;
let prefix -- a b = int_of_float(float_of_int(a) - float_of_int(b)) ;;
let prefix ** a b = int_of_float(float_of_int(a) * float_of_int(b)) ;;
let prefix ^^ a b = int_of_float( power (float_of_int(a))
                                        (float_of_int(b)));;
let prefix ^^^^ a b = power a b;;
let prefix // a b = int_of_float(float_of_int(a) / float_of_int(b)) ;;
```

On définit un type enregistrement $options$ pour prendre en compte les choix de l'utilisateur. On propose une option $auto$ de manière à représenter une fonction quelconque simplement.

```
type options = {scallingAuto : bool ;quad : bool ;lineMode : bool; xmin :
float; xmax : float; ymin : float; ymax : float; color : color };;

let auto = {scallingAuto = true; quad = true; lineMode = true; xmin = 0. ;
xmax = 0.; ymin = 0. ; ymax = 0.; color = blue };;
```

On aura besoin de la valeur de π obtenue de la manière suivante :

```
let pi = acos(-1.);;
```

On programme un certain nombre de fonctions auxiliaires :

```
let drawNum x =
match x with
|x when x = 0. -> draw_string("0")
|x when mod_float x pi = 0. -> draw_string((string_of_int((int_of_float(x /
pi))))^"pi")
|x when float_of_int(int_of_float(x)) = x ->
draw_string(string_of_int(int_of_float(x)))
|_ -> draw_string(string_of_float(x));;
```

La fonction $drawNum$ sera utilisée pour afficher des nombres dans la fenêtre graphique qui prend comme argument un flottant. De sorte qu'un entier de type $float$ soit affiché sans virgule flottante. On ajoute aussi les multiples de π de sorte que $x \times Pi$ soit affiché au lieu de sa valeur numérique.

```
let moins x =
(0 -- x);;
```

La fonction $moins$ est une fonction obsolète qui renvoie l'opposé d'un nombre de type int.

```
let abs_or_nul x =
match x with
|x when x < 0. -> -x
|x when x > 0. -> 0.
|_ -> x;;
```

On programme une fonction analogue à abs_float qui, pour un flottant x passé en argument, renvoie :

$$\begin{cases} |x| \ si \ x < 0 \\ 0 \quad si \ x > 0 \end{cases}$$

On programme ensuite la fonction $graph()$. Pour ne pas surcharger le type $options$, on décide de passer en argument les graduations des échelles :

$$graph : float\ vect\ vect \rightarrow largeur\ de\ la\ fenêtre \rightarrow hauteur\ de\ la\ fenêtre \rightarrow espacement\ des\ graduations\ verticales \rightarrow espacement\ des\ graduations\ horizontales \rightarrow options \rightarrow unit$$

```
let graph vect x y sx sy options=
```

Pour ouvrir la fenêtre de la taille souhaitée, il suffit de créer une chaine en concaténant la chaîne de caractère issue de l'entier $'x'$," × ", et l'entier 'y' converti en chaîne. Puis d'ouvrir normalement la fenêtre ainsi définie.

```
let fenetre = string_of_int(x)^"x"^string_of_int(y) in
  open_graph fenetre;
```

On définit quatre variables qui auront leurs utilités dans l'évaluation d'une échelle de représentation adaptée. $xmin$ et $xmax$ représentent les bornes de l'intervalle d'étude. $ymin$ et $ymax$ représentent les ordonnées maximales et minimales souhaitées pour la représentation. Si l'option $auto$ est passée en argument, il faudra définir ces variables automatiquement de manière à représenter les coordonnées successives enregistrées dans une matrice dans leurs intégralités. On doit donc scanner la matrice pour obtenir les valeurs maximums et minimums prises par les abscisses et les ordonnées.

```
  let xmin = ref vect.(0).(0) and xmax = ref 0.
  and ymin = ref 0. and ymax = ref 0. in
    if options.scallingAuto
    then
      begin
      let n = ref vect.(0).(1) and  m = ref vect.(0).(0) in
        for i=0 to ((vect_length vect) -- 100) do
          n := vect.(i).(1);
          m := vect.(i).(0);
          if !n >= !ymax then ymax := !n;
          if !n <= !ymin then ymin := !n;
          if !m >= !xmax then xmax := !m;
        done;
    end
    else
      begin
      xmin := options.xmin ;xmax := options.xmax;
      ymin := options.ymin ;ymax := options.ymax;
      end;
```

On peut alors créer nos échelles de représentation (échelle verticale et horizontale).

```
(*Définition de l'échelle*)
    let scalling_x = float_of_int(x) / (!xmax - !xmin)
    and scalling_y = float_of_int(y) / (!ymax - !ymin) in
```

La matrice associée à la fonction est alors aisément affichable de manière continu ou point par point.

```
    (*Traçage du vecteur*)
    set_color options.color;
    if options.lineMode
    then
    (*LineMode, Traçage par ligne*)
      begin
        moveto (int_of_float(((vect.(0).(0) -(!xmin)) * scalling_x)))
               (int_of_float(((vect.(0).(1) -(!ymin)) * scalling_y)));
        for i=1 to ((vect_length vect) -- 1) do
        (*Correctif pour les vecteurs se finissant par (0,0)*)
        if (((int_of_float(((vect.(i).(0) -(!xmin)) * scalling_x))) <> 0)
&&
           ((int_of_float(((vect.(i).(1) -(!ymin)) * scalling_y))) <> 0))
        then begin
        lineto (int_of_float(((vect.(i).(0) -(!xmin)) * scalling_x)))
               (int_of_float(((vect.(i).(1) -(!ymin)) * scalling_y)));
             end
        done;
      end
    else
    (*Traçage par point*)
      begin
        for i=0 to ((vect_length vect) -- 1) do
        plot  (int_of_float(((vect.(i).(0) -(!xmin)) * scalling_x)))
              (int_of_float(((vect.(i).(1) -(!ymin)) * scalling_y)));
        done;
      end;
```

Il ne reste plus qu'à afficher les axes et à les graduer selon l'échelle de la représentation :

```
    set_color black;
    (*Affichage des axes*)
      moveto (0) (int_of_float((- !ymin) * (scalling_y)));
      lineto (x) (int_of_float((- !ymin) * (scalling_y)));
      moveto (int_of_float((abs_or_nul !xmin) * (scalling_x))) (0);
      lineto (int_of_float((abs_or_nul !xmin) * (scalling_x))) (y);
    (* Graduation des axes *)
    for i= int_of_float(!xmin / sx - 1.) to int_of_float(!xmax / sx + 1.)
do
      moveto (int_of_float((float_of_int(i) * sx  - !xmin) * (scalling_x)))
(int_of_float(((- !ymin)* (scalling_y) + 0.)));
      lineto (int_of_float((float_of_int(i) * sx  - !xmin) * (scalling_x)))
(int_of_float(((- !ymin)* (scalling_y) + 5.)));
      drawNum (float_of_int(i)*sx);
    done;
    for i= int_of_float(!ymin / sy - 1.)  to int_of_float(!ymax / sy + 1.)
do
      moveto (int_of_float(((abs_or_nul !xmin)* (scalling_x) + 0.)))
(int_of_float((float_of_int(i) * sy - !ymin)* (scalling_y)));
      lineto (int_of_float(((abs_or_nul !xmin)* (scalling_x) + 5.)))
(int_of_float((float_of_int(i) * sy - !ymin)* (scalling_y)));
      drawNum (float_of_int(i)*sy);
    done;
```

On rend possible, selon les options choisies par l'utilisateur, de quadriller la fenêtre graphique :

```
    (*Affichage du quadrillage*)
    if options.quad then
      begin
        for i = 0 to x do
          for j = (int_of_float(!ymin) -- 1)  to int_of_float(!ymax / sy +
1.)  do
          plot (i ++ i mod 2) (int_of_float((float_of_int(j) * sy - !ymin)*
(scalling_y)));
          done;
        done;
        for i = 0 to y do
          for j = (int_of_float(!xmin) -- 1) to int_of_float(!xmax /  sx +
1.) do
          plot (int_of_float((float_of_int(j) * sx - !xmin)* (scalling_x)))
(i ++ i mod 2);
          done;
        done;
      end
;;
```

On est, désormais, en mesure de tracer toutes matrices de coordonnées associées à la courbe représentative d'une fonction quelconque. Il nous reste donc à définir des fonctions informatiques de « conversions », qui à toute fonction mathématique, associent une matrice des coordonnées de sa courbe représentative. On retrouvera la fonction *create_vect* déjà connue.

```
(*Création d'une matrice de taille adaptée*)
let create_vect xo xf dx = make_matrix (int_of_float((xf-xo)/dx + 1./dx)) 2
0.;;
```

Cette transformation se fait très simplement de manière itérative en calculant toutes les images sur un intervalle d'étude donné en fonction d'un pas d'itération. On définira de même une fonction analogue pour associer une matrice à la courbe représentative d'une fonction polaire.

```
(*Vectorisation des valeurs d'une fonction quelconque*)
let fonction_to_vect f xo xf dx =
  let i = ref xo and ii = ref 0
    and vect = (create_vect xo xf dx) in
      let y = ref (f(xo)) in
          while !i <= xf do
          (vect.(!ii).(0) <- !i);
          (vect.(!ii).(1) <- !y);
          y  := f(!i);
          i := !i + dx;
          ii:= (int_of_float(float_of_int(!ii)+ 1.));
          done;
      vect;;
```

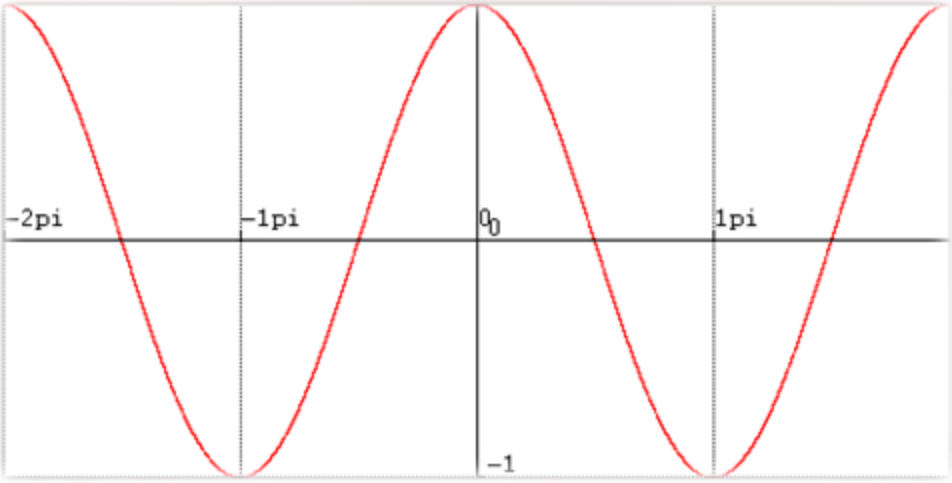
```
(*Vectorisation des valeurs d'une courbe polaire r=f(T)*)
let polaire_to_vect f t0 tf dt =
  let t = ref t0 and tt = ref 0
    and vect = (create_vect t0 tf dt) in
      let r = ref (f(t0)) in
          while !t <= tf do
          (vect.(!tt).(0) <- (!r * cos(!t)));
          (vect.(!tt).(1) <- (!r * sin(!t)));
          r := f(!t);
```

```
            t := !t + dt;
            tt:= (int_of_float(float_of_int(!tt)+ 1.));
            done;
        vect;;
```

Exemples d'utilisations :

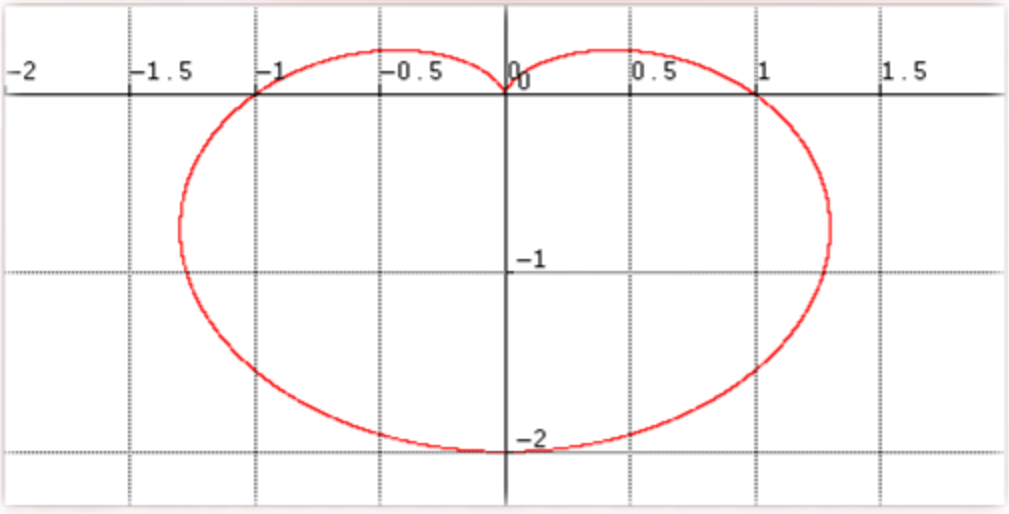
```
let perso = {scallingAuto = false ; quad = true; lineMode = false ;xmin = -
2. * pi ; xmax = 2. * pi ; ymin = -1. ; ymax = 1.; color = red};;

let f = function x -> cos x;;

graph (fonction_to_vect f (-2. * pi) (2. * pi) 0.001) 500 250 (pi) 1.
perso;;
```

Courbe représentative de $f(x) = \cos(x)$




```
let perso = {scallingAuto = false ; quad = true; lineMode = false ;xmin = -
2.; xmax = 2.; ymin = -2.3 ; ymax = 0.5; color = red};;

let f = function t -> (1. - sin t);;

graph (polaire_to_vect f 0. (2. * pi) 0.001) 500 250 (0.5) 1. perso;;
```

Courbe représentative de $\rho = 1 - \sin\theta$



Sources

Source 1 :

Méthode d'euler.ml

```
#open "float";;

(*Création d'une matrice de taille adaptée*)
let create_vect xo xf dx = make_matrix (int_of_float((xf-xo)/dx + 1./dx +
dx)) 2 0.;;

(*Euler 1 - récursive*)

let rec euler1 f' yo xo xf dx =
  let y1 = yo + f'(xo,yo) * dx in
  match xo with
  |xo when xo < xf  -> (xo,yo)::(euler1 f' y1 (xo + dx) xf dx);
  |_                -> []
;;

(*Euler 1 - itérative*)

let euler1 f' yo xo xf dx =
  let i = ref xo and ii = ref 0
    and vect = (create_vect xo xf dx) in
      let y = ref yo in
          while !i <= xf do
          (vect.(!ii).(0) <- !i);
          (vect.(!ii).(1) <- !y);
          y  := !y + f'(!i,!y) * dx;
          i := !i + dx;
          incr ii;
        done;
      vect;;

(*Euler 2*)

let euler2 f'' y'o yo xo xf dx =
  let y'' = ref (f''(xo,yo,y'o)) and y' = ref y'o and y = ref yo in
    let i = ref xo and ii = ref 0
    and vect = (create_vect xo xf dx) in
      while !i <= xf do
        (vect.(!ii).(0) <- !i);
        (vect.(!ii).(1) <- !y);
        y   := !y + dx * ( !y' + ( !y'' * dx));
        y'  := !y' + !y'' * dx;
        y'' := f''(!i,!y,!y');
        i := !i + dx;
        incr ii;
      done;
vect;;

(*
  - Version finale (Copie de Méthode d'Euler 8.ml)
*)
```

Source 2 :

Méthode de Runge Kutta.ml

```
#open "float";;

(*Création d'une matrice de taille adaptée*)
let create_vect xo xf dx = make_matrix (int_of_float((xf-xo)/dx + 1./dx +
dx)) 2 0.;;

(* Runge Kutta 4 Equa-diff Ordre 1*)

let RK4_1 f' yo xo xf dx =
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0. in
  let x = ref xo and y = ref yo in
  let xx = ref 0 and vect = (create_vect xo xf dx) in
    while !x <= xf do
      (vect.(!xx).(0) <- !x);
      (vect.(!xx).(1) <- !y);
      K1 := f'(!x,!y);
      K2 := f'(!x + 0.5 * dx , !y + 0.5 * !K1 * dx);
      K3 := f'(!x + 0.5 * dx , !y + 0.5 * !K2 * dx);
      K4 := f'(!x + 0.5 , !y + !K3 * dx);
      y := !y + (!K1 + 2. * !K2 + 2. * !K3 + !K4) * dx / 6. ;
      x := !x + dx;
      incr xx;
    done;
vect;;


(* Runge Kutta 4 Equa-diff Ordre 2*)

let RK4_2 f'' y'o yo xo xf dx =
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0.
  and J1 = ref 0. and J2 = ref 0. and J3 = ref 0. and J4 = ref 0.
  and x = ref xo and y = ref yo and y' = ref y'o in
  let xx = ref 0 and vect = (create_vect xo xf dx) in
    while !x <= xf do
      (vect.(!xx).(0) <- !x);
      (vect.(!xx).(1) <- !y);
      J1 := !y'*dx;
      K1 := f''(!x, !y, !y')*dx;
      J2 := (!y' + !K1 / 2.)*dx;
      K2 := f''(!x,( !y + !J1 / 2.),(!y' + !K1 / 2.))*dx;
      J3 := (!y' + !K2 / 2.)*dx;
      K3 := f''(!x, (!y + !J2 / 2.), (!y' + !K2 / 2.))*dx;
      J4 := (!y' + !K3)*dx;
      K4 := f''(!x, (!y + !J3), (!y' + !K3))*dx;
      y' := !y' + (!K1 + 2. * !K2 + 2. * !K3 + !K4)/ 6. ;
      y  := !y  + (!J1 + 2. * !J2 + 2. * !J3 + !J4)/ 6. ;
      x  := !x + dx;
      incr xx;
    done;
vect;;

(*
  - Version finale (Copie de Runge Kutta 4.ml)
*)
```

Source 3 :

Méthode de Runge Kutta F 5.ml

```
#open "float";;

(*Création d'une matrice de taille adaptée*)
let create_vect xo xf dx = make_matrix (int_of_float((xf-xo)/dx + 1./dx +
dx)) 2 0.;;

(* Runge Kutta F 5 Equa-diff Ordre 1*)

let RKF5_1 f' yo xo xf dx =
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0. and K5 =
ref 0. and K6 = ref 0.in
  let x = ref xo and y = ref yo in
  let xx = ref 0 and vect = (create_vect xo xf dx) in
    while !x <= xf do
      (vect.(!xx).(0) <- !x);
      (vect.(!xx).(1) <- !y);
      K1 := dx * f'(!x,!y);
      K2 := dx * f'(!x + (1./4.) * dx, !y + (1./4.) * !K1);
      K3 := dx * f'(!x + (3./8.) * dx, !y + (3./32.) * !K1 + (9./32.) *
!K2);
      K4 := dx * f'(!x + (12./13.)*dx, !y + (1932./2197.) * !K1 -
(7200./2197.) * ! K2 + (7296./2197.) * !K3);
      K5 := dx * f'(!x + dx          , !y + (439./216.) * !K1 - (8.) * !K2
+ (3680./513.) * !K3 - (845./4104.) * !K4 );
      K6 := dx * f'(!x + (1./2.) * dx, !y - (8./21.) * !K1 + 2. * !K2 -
(3544./2565.) * !K3 + (1859./4104.) * !K4 - (11./40.) * !K5);
      y := !y + (16./135.) * !K1 + (6656./12825.)* !K3 + (28561./56430.) *
!K4 - (9./50.) * !K5 + (2./55.) * !K6;
      x := !x + dx;
      xx := (int_of_float(float_of_int(!xx)+ 1.));
    done;
vect;;
```

Source 4 :

Représentation Graphique.ml

```
#open "float";;

(*Fonction "plot" améliorée*)

#open "graphics";;

type options = {scallingAuto : bool ;quad : bool ;lineMode : bool; xmin :
float; xmax : float; ymin : float; ymax : float; color : color };;
let auto = {scallingAuto = true; quad = true; lineMode = true; xmin = 0. ;
xmax = 0.; ymin = 0. ; ymax = 0.; color = blue };;


let prefix ++ a b = int_of_float(float_of_int(a) + float_of_int(b)) ;;
let prefix -- a b = int_of_float(float_of_int(a) - float_of_int(b)) ;;
let prefix ** a b = int_of_float(float_of_int(a) * float_of_int(b)) ;;
let prefix ^^ a b = int_of_float( power (float_of_int(a))
(float_of_int(b)));;
let prefix ^^^^ a b = power a b;;
let prefix // a b = int_of_float(float_of_int(a) / float_of_int(b)) ;;

let pi = acos(-1.);;

let drawNum x =
match x with
|x when x = 0. -> draw_string("0")
|x when mod_float x pi = 0. -> draw_string((string_of_int((int_of_float(x /
pi))))^"pi")
|x when float_of_int(int_of_float(x)) = x ->
draw_string(string_of_int(int_of_float(x)))
|_ -> draw_string(string_of_float(x));;


let moins x =
(0 -- x);;


let abs_or_nul x =
match x with
|x when x < 0. -> -x
|x when x > 0. -> 0.
|_ -> x;;

let graph vect x y sx sy options=
  let fenetre = string_of_int(x)^"x"^string_of_int(y) in
  open_graph fenetre;
  (*clear_graph();*)
  let xmin = ref vect.(0).(0) and xmax = ref 0.
  and ymin = ref 0. and ymax = ref 0. in
    if options.scallingAuto
    then
      begin
      let n = ref vect.(0).(1) and  m = ref vect.(0).(0) in
        for i=0 to ((vect_length vect) -- 100) do
          n := vect.(i).(1);
          m := vect.(i).(0);
          if !n >= !ymax then ymax := !n;
          if !n <= !ymin then ymin := !n;
```

```
          if !m >= !xmax then xmax := !m;
        done;
    end
    else
      begin
      xmin := options.xmin ;xmax := options.xmax;
      ymin := options.ymin ;ymax := options.ymax;
      end;
    (*Définition de l'échelle*)
    let scalling_x = float_of_int(x) / (!xmax - !xmin) and scalling_y =
float_of_int(y) / (!ymax - !ymin) in
    (*Traçage du vecteur*)
    set_color options.color;
    if options.lineMode
    then
    (*LineMode, Traçage par ligne*)
      begin
        moveto (int_of_float(((vect.(0).(0) -(!xmin)) * scalling_x)))
               (int_of_float(((vect.(0).(1) -(!ymin)) * scalling_y)));
        for i=1 to ((vect_length vect) -- 1) do
        (*Correctif pour les vecteurs se finissant par (0,0)*)
        if (((int_of_float(((vect.(i).(0) -(!xmin)) * scalling_x))) <> 0)
&&
           ((int_of_float(((vect.(i).(1) -(!ymin)) * scalling_y))) <> 0))
        then begin
        lineto (int_of_float(((vect.(i).(0) -(!xmin)) * scalling_x)))
               (int_of_float(((vect.(i).(1) -(!ymin)) * scalling_y)));
             end
        done;
      end
    else
    (*Traçage par point*)
      begin
        for i=0 to ((vect_length vect) -- 1) do
        plot  (int_of_float(((vect.(i).(0) -(!xmin)) * scalling_x)))
              (int_of_float(((vect.(i).(1) -(!ymin)) * scalling_y)));
        done;
      end;
    set_color black;
    (*Affichage des axes*)
      moveto (0) (int_of_float((- !ymin) * (scalling_y)));
      lineto (x) (int_of_float((- !ymin) * (scalling_y)));
      moveto (int_of_float((abs_or_nul !xmin) * (scalling_x))) (0);
      lineto (int_of_float((abs_or_nul !xmin) * (scalling_x))) (y);
    (* Graduation des axes *)
    for i= int_of_float(!xmin / sx - 1.) to int_of_float(!xmax / sx + 1.)
do
      moveto (int_of_float((float_of_int(i) * sx  - !xmin) * (scalling_x)))
(int_of_float(((- !ymin)* (scalling_y) + 0.)));
      lineto (int_of_float((float_of_int(i) * sx  - !xmin) * (scalling_x)))
(int_of_float(((- !ymin)* (scalling_y) + 5.)));
      drawNum (float_of_int(i)*sx);
    done;
    for i= int_of_float(!ymin / sy - 1.)  to int_of_float(!ymax / sy + 1.)
do
      moveto (int_of_float(((abs_or_nul !xmin)* (scalling_x) + 0.)))
(int_of_float((float_of_int(i) * sy - !ymin)* (scalling_y)));
      lineto (int_of_float(((abs_or_nul !xmin)* (scalling_x) + 5.)))
(int_of_float((float_of_int(i) * sy - !ymin)* (scalling_y)));
      drawNum (float_of_int(i)*sy);
    done;
```

```
    (*Affichage du quadrillage*)
    if options.quad then
      begin
        for i = 0 to x do
          for j = (int_of_float(!ymin) -- 1)  to int_of_float(!ymax / sy +
1.)  do
          plot (i ++ i mod 2) (int_of_float((float_of_int(j) * sy - !ymin)*
(scalling_y)));
          done;
        done;
        for i = 0 to y do
          for j = (int_of_float(!xmin) -- 1) to int_of_float(!xmax /  sx +
1.) do
          plot (int_of_float((float_of_int(j) * sx - !xmin)* (scalling_x)))
(i ++ i mod 2);
          done;
        done;
      end
;;

(*Création d'une matrice de taille adaptée*)
let create_vect xo xf dx = make_matrix (int_of_float((xf-xo)/dx + 1./dx)) 2
0.;;

(*Vectorisation des valeurs d'une fonction quelconque*)
let fonction_to_vect f xo xf dx =
  let i = ref xo and ii = ref 0
    and vect = (create_vect xo xf dx) in
      let y = ref (f(xo)) in
          while !i <= xf do
          (vect.(!ii).(0) <- !i);
          (vect.(!ii).(1) <- !y);
          y  := f(!i);
          i := !i + dx;
          ii:= (int_of_float(float_of_int(!ii)+ 1.));
          done;
      vect;;


(*Vectorisation des valeurs d'une courbe polaire r=f(T)*)
let polaire_to_vect f t0 tf dt =
  let t = ref t0 and tt = ref 0
    and vect = (create_vect t0 tf dt) in
      let r = ref (f(t0)) in
          while !t <= tf do
          (vect.(!tt).(0) <- (!r * cos(!t)));
          (vect.(!tt).(1) <- (!r * sin(!t)));
          r := f(!t);
          t := !t + dt;
          tt:= (int_of_float(float_of_int(!tt)+ 1.));
          done;
      vect;;

(*
  - Version finale basée sur Représentation Graphique 2.ml.
*)
```

Source 5 :

Résolution numérique.ml

```
#open "float";;

directory "F:\TIPE\Source\Résolution Numérique";;
include "Méthode d'Euler.ml";;
include "Runge Kutta.ml";;
include "Représentation Graphique.ml";;


(*Décharge d'un condensateur (U(x))*)

let RC = 80.;;

let f'(x,y) =
- ( 1. / RC )* y;;

let numerique = {scallingAuto = false ; quad = true; lineMode = true ;
xmin = 0. ; xmax = 200. ; ymin = 0. ; ymax = 6.; color = red};;
let numerique2 = {scallingAuto = false ; quad = true; lineMode = true ;
xmin = 0. ; xmax = 200. ; ymin = 0. ; ymax = 6.; color = blue};;
let analytique = {scallingAuto = false ; quad = true; lineMode = false ;
xmin = 0. ; xmax = 200. ; ymin = 0. ; ymax = 6.; color = green};;

graph (euler1 f' 6. (0.) 400. 10.) 500 250 30. 1. numerique;;
graph (RK4_1 f' 6.0 0. 400. 10.) 500 250 30. 1. numerique2;;
graph (fonction_to_vect (fun t -> 6. * exp(-t/RC)) 1. 400. 0.1) 500 250 30.
1. analytique;;



(* Fonction Exponentielle*)

let exponentiel(x,y) =
y ;;

graph (euler1 exponentiel 2.71 1. 3. 0.01) 500 250 1. 10. auto;;
graph (RK4_1  exponentiel 2.71 1. 3. 0.001) 500 250 1. 1. auto;;
graph (fonction_to_vect (fun x -> exp(x)) 1. 3. 0.001) 500 250 1. 10.
auto;;

(*Systeme masse ressort avec frottement fluide*)

let numerique = {scallingAuto = false ; quad= true; lineMode = true ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = red};;
let analytique = {scallingAuto = false ; quad = true; lineMode = false ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = green};;
let numerique2 = {scallingAuto = false ; quad= true; lineMode = true ;
xmin = 0. ; xmax = 500. ; ymin = -5. ; ymax = 5.; color = blue};;

let a = 5. and m = 500. and k = 10.;;

let solution t =
let racineDelta = sqrt( 4. * k / m - (a ^^^^ 2.) / (m ^^^^ 2.)) in
exp( - a * t / (2.*m))*(5.*cos(racineDelta * t /2. ));;


let f''1(x,y,y') =
  - ( a / m ) * y' - ( k / m ) * y ;;
```

```
graph (euler2 f''1 0. 5. 0. 500. 3.) 500 250 100. 1. numerique;;
graph (RK4_2 f''1 0. 5. 0. 500. 3.) 500 250 100. 1. numerique2;;
graph (fonction_to_vect solution 0. 500. 0.01) 500 250 100. 1. analytique;;


(* Exemple à deux variables*)

let f''(x,y,y') =
3. * exp( - x ) - 0.4 * y;;

graph (euler2 f'' 0. 5. (0.) 10. 0.0001) 500 250 1. 1. auto;;
graph (RK4_2 f'' 0. 5. (0.) 10. 0.0001) 500 250 1. 1. auto;;


(*Fonction quelconque*)

let perso = {scallingAuto = false ; quad = true; lineMode = false ;xmin = -
2. * pi ; xmax = 2. * pi ; ymin = -1. ; ymax = 1.; color = red};;

let f = function x -> cos x;;

graph (fonction_to_vect f (-2. * pi) (2. * pi) 0.001) 500 250 (pi) 1.
perso;;

let perso = {scallingAuto = false ; quad = true; lineMode = false ;xmin = -
2.; xmax = 2.; ymin = -2.3 ; ymax = 0.5; color = red};;

let f = function t -> (1. - sin t);;

graph (polaire_to_vect f 0. (3. * pi) 0.001) 500 250 (0.5) 1. perso;;


(* Exemple 3 *)
let ex3'(x,y) =
-tan(x) * y;;

graph (euler1 ex3' 1. (0.) 10. 0.001) 500 250 pi 1. perso;;

(*
  - Basé sur Méthode d'Euler.ml.
  - Basé sur Runge Kutta.ml.
  - Basé sur Représentation Graphique.ml.
*)
```

Source 6 :

Problème à deux corps mode continu.ml

```
#open "graphics";;
#open "float";;
#open "format";;

(*Opérations sur les entiers*)

  let prefix ++ a b = int_of_float(float_of_int(a) + float_of_int(b)) ;;
  let prefix -- a b = int_of_float(float_of_int(a) - float_of_int(b)) ;;
  let prefix ** a b = int_of_float(float_of_int(a) * float_of_int(b)) ;;
  let prefix ^^ a b = int_of_float( power (float_of_int(a))
(float_of_int(b)));;
  let prefix // a b = int_of_float(float_of_int(a) / float_of_int(b)) ;;

(*Opération sur les flottants*)
  (* '**.' reste disponible.*)
  let prefix ^^^^ a b = power a b;;

(*Référencement d'un couple*)
  let prefix ~:= (a,b) (c,d) = a := c;  b := d;;

(*Conversions des types*)
  let int(a) = int_of_float a;;
  let float(a) = float_of_int a;;
  let string(a) = string_of_float a;;

(*Déclarations des types.*)
  type corps   =
    {mutable masse    : float;
    mutable rayon     : int;
    mutable coord_x   : float;
    mutable coord_y   : float;
    mutable vitesse_x : float;
    mutable vitesse_y : float;
    };;

  type options =
    {mutable dt       : float;
    mutable zoom      : float;
    mutable origine_x : int ;
    mutable origine_y : int ;
    mutable scale_F   : float;
    mutable scale_v   : float;
    mutable dispTime  : bool;
    mutable dispState : bool;
    mutable dispTraj  : bool;
    mutable dispM     : bool;
    mutable dispG     : bool;
    mutable dispCorps : bool;
    mutable blinker   : bool;
    mutable hypermode : bool;
    mutable lineto    : bool;
    mutable reset     : bool;
    mutable elag      : bool;
  };;
```

```
  type state  =
    {mutable corps1  : corps;
    mutable corps2   : corps;
    mutable partFict : corps;
    mutable centreG  : corps;
    mutable F        : float;
  };;

(*Déclaration d'un printer pour le type Corps*)

  let pp_print_corps state c =
    pp_print_string state ("Masse      : "^(string c.masse));
    print_string ("\n Rayon      : "^(string_of_int c.rayon));
    print_string ("\n Coordonées : ("^(string c.coord_x)^", "^(string
c.coord_y)^")");
    print_string ("\n Vitesse    : ("^(string c.vitesse_x)^", "^(string
c.vitesse_y)^")");
  ;;

  let print_corps = pp_print_corps std_formatter;;

  install_printer "print_corps";;


(*Couleurs*)
  let marron  = (rgb 135 085 065);;
  let rouge   = (rgb 200 000 000);;
  let bleu    = (rgb 000 000 200);;
  let vert    = (rgb 000 200 000);;
  let c_oie   = (rgb 219 219 000);;

(*Reset key_pressed (Type unit)*)
  let readkey() = let c = ref `a` in c:= read_key();;

(*Pause jusqu'à ce qu'une touche soit pressée*)
  let wait() = while not(key_pressed()) do done; readkey() ;;

(*Pause de x ms*)
  let pause x = sound 20000 x;;

(*Exceptions*)
  exception collision;;

(*Auxiliaire - Utilisé dans le test pour les trajectoires.*)
let gt1 a = if (a < 1.) then 1 else int a;;

(*Affichage de l'état de la simulation*)
  let vectString_of_state state options =
  [|
    "Vitesse de simulation : "^(string options.dt);
    "Zoom : x"^(string options.zoom);
    "Echelle Vitesse : 1/"^(string (options.scale_v))^" ème";
    "Echelle Force :   1/"^(string (options.scale_F))^" ème";
    "";
    "Corps 1 :";
    "Xa    : "^(string state.corps1.coord_x);
    "Ya    : "^(string state.corps1.coord_y);
    "Vxa   : "^(string state.corps1.vitesse_x)^" m/s";
    "Vya   : "^(string state.corps1.vitesse_y)^" m/s";
    "|Va|  : "^(string (sqrt((state.corps1.vitesse_x^^^^2.) +
(state.corps1.vitesse_y^^^^2.))))^" m/s";
```

```
    "";
    "Corps 2 :";
    "Xb    : "^(string state.corps2.coord_x);
    "Yb    : "^(string state.corps2.coord_y);
    "Vxb   : "^(string state.corps2.vitesse_x)^" m/s";
    "Vyb   : "^(string state.corps2.vitesse_y)^" m/s";
    "|Vb|  : "^(string (sqrt((state.corps2.vitesse_x^^^^2.) +
(state.corps2.vitesse_y^^^^2.))))^" m/s";
    "";
    "Force d'interaction (De 1 sur 2) :";
    "F     :"^(string (state.F))^" N";
    "";
    "Centre de Masse G";
    "Xg    : "^(string state.centreG.coord_x);
    "Yg    : "^(string state.centreG.coord_y);
    "Vxg   : "^(string state.centreG.vitesse_x)^" m/s";
    "Vyg   : "^(string state.centreG.vitesse_y)^" m/s";
    "|Vg|  : "^(string (sqrt((state.centreG.vitesse_x^^^^2.) +
(state.centreG.vitesse_y^^^^2.))))^" m/s";
    "";
    "Particule Fictive M";
    "Xm    : "^(string state.partFict.coord_x);
    "Ym    : "^(string state.partFict.coord_y);
    "Vxm   : "^(string state.partFict.vitesse_x)^" m/s";
    "Vym   : "^(string state.partFict.vitesse_y)^" m/s";
    "|Vm|  : "^(string (sqrt((state.partFict.vitesse_x^^^^2.) +
(state.partFict.vitesse_y^^^^2.))))^" m/s";

  |];;

(*Messages d'informations*)
  let infos =
    [|
    (*0*) [|"Origine décalée : ";" Y = y + 1"|];
    (*1*) [|"Origine décalée : ";" X = x - 1"|];
    (*2*) [|"Origine décalée : ";" Y = y - 1"|];
    (*3*) [|"Origine décalée : ";" X = x + 1"|];
    (*4*) [|"Zoom - : (divisé par 1.2)";""|];
    (*5*) [|"Zoom + : (multiplié par 1.2)";""|];
    (*6*) [|"Vitesse de simulation doublée :";"Attention les erreurs
s'accumulent."|];
    (*7*) [|"Vitesse de simulation réduite :";"Attention les erreurs
s'accumulent."|];
    (*8*) [|"Affichage de l'état de la simulation :";"Ralentit la
simulation."|];
    (*9*) [|"Affichage de la particule fictive :";"Purement théorique."|];
    (*10*)[|"Affichage de centre de masse :";"Purement théorique."|];
    (*11*)[|"Affichage des trajectoires :";"Ralentit progressivement la
simulation."|];
    (*12*)[|"Affichage du temps : ";"Unité : Seconde."|];
    (*13*)[|"Sens normal de simulation :";"Attention les erreurs
s'accumulent."|];
    (*14*)[|"Sens inverse de simulation :";"Attention les erreurs
s'accumulent."|];
    (*15*)[|"Mode Anti-Blinker : Ralentit";"Grandement la vitesse de
simulation."|];
    (*16*)[|"Hypermode, désactive les options :";"Accélère la
simulation."|];
    (*17*)[|"En Pause : ";"Appuyer sur une touche pour continuer"|];
    (*18*)[|"Affichage des trajectoires: ";"Mode continu"|];
```

```
    (*19*)[|"Trajectoires (listes): ";"Perte d'une information sur
deux."|];
    (*20*)[|"Trajectoires (listes): ";"Remise à zero"|];
    |];;

(*Fonction d'affichage du temps*)
  let draw_time time =
    (*Nettoyage de la zone d'écriture*)
    set_color white;
    fill_rect (size_x()-- 300) (size_y()-- 20) 300 20;
    (*Ecriture*)
    set_color black;
    moveto (size_x()-- 300) (size_y()-- 20);
    draw_string ("Temps  : "^(string_of_float time)^" s.");
  ;;

(*Affichage de l'état de la simulation*)
let draw_state states =
    let n = vect_length states -- 1 in
    (*Nettoyage de la zone d'écriture*)
      set_color white;
      fill_rect (size_x()-- 300) (40) 300 (size_y()-- 60);
    (*Ecriture*)
      set_color black;
      for i=0 to n do
        moveto (size_x()-- 300) (size_y() -- (15**i ++ 35)) ;
        draw_string states.(i);
      done;
;;

(*Fonction d'affichage des messages de notifications.*)
  let notification message =
    (*Nettoyage de la zone d'écriture*)
    set_color white;
    fill_rect (size_x()-- 300) 0 300 40;
    (*Ecriture*)
    set_color black;
    (*Ligne 1*)
    moveto (size_x()-- 300) 20 ;
    draw_string message.(0);
    (*Ligne 2*)
    moveto (size_x()-- 300) 0 ;
    draw_string message.(1);
  ;;


(*Fonction d'interaction clavier.*)
  let waitif options state=
    if key_pressed()
      then
        begin
        match read_key() with
        (*Déplacement*)
        | `z`   -> clear_graph(); (notification infos.(0));
options.origine_y <- options.origine_y ++ 1;options;
        | `q`   -> clear_graph(); (notification infos.(1));
options.origine_x <- options.origine_x -- 1;options;
        | `s`   -> clear_graph(); (notification infos.(2));
options.origine_y <- options.origine_y -- 1;options;
        | `d`   -> clear_graph(); (notification infos.(3));
options.origine_x <- options.origine_x ++ 1;options;
```

```
        (*Zoom + et Zoom -*)
        | `-`   -> clear_graph(); (notification infos.(4)); options.zoom <-
options.zoom / 1.2;options;
        | `+`   -> clear_graph(); (notification infos.(5)); options.zoom <-
options.zoom * 1.2;options;
        (*Vitesse de simulation*)
        | `r`   -> (notification infos.(6)); options.dt <- options.dt * 2.;
options;
        | `l`   -> (notification infos.(7)); options.dt <- options.dt / 2.;
options;
        (*Information*)
        | `i`   -> clear_graph(); (notification infos.(8));
options.dispState <- not(options.dispState); options;
        (*Affichage de la Particule Fictive*)
        | `m`   -> clear_graph(); (notification infos.(9)); options.dispM
<- not(options.dispM) ;options;
        (*Affichage du Centre de Masse G*)
        | `g`   -> clear_graph(); (notification infos.(10)); options.dispG
<- not(options.dispG) ;options;
        (*Affichage des trajectoires*)
        | `a`   -> clear_graph(); (notification infos.(11));
options.dispTraj <- not(options.dispTraj); options;
        (*Mode "Lineto" sur les trajectoires*)
        | `c`   -> clear_graph(); (notification infos.(18)); options.lineto
<- not(options.lineto); options;
        (*Elagage des listes*)
        | `e`   -> clear_graph(); (notification infos.(19)); options.elag
<- true; options;
        (*Reset listes*)
        | `x`   -> clear_graph(); (notification infos.(20)); options.reset
<- true; options;
        (*Affichage du temps*)
        | `t`   -> clear_graph(); (notification infos.(12));
options.dispTime <- not(options.dispTime ); options;
        (*Sens de la simulation*)
        | `>`   -> (notification infos.(13)); options.dt <-  abs_float
options.dt; options;
        | `<`   -> (notification infos.(14)); options.dt <- - abs_float
options.dt; options;
        (*Anti-blinker - réduit le clignotement*)
        | `b`   -> (notification infos.(15)); options.blinker <-
not(options.blinker); options;
        (*Hypermode*)
        | `h`   -> clear_graph(); (notification infos.(16));
options.hypermode <- not(options.hypermode);  options;
        (*Exit*)
        | `\027`-> close_graph(); raise Exit; options;
        (*Pause*)
        |   _   -> (notification infos.(17)); draw_state
(vectString_of_state state options); wait(); clear_graph(); options;
      end
    else options;;
```

```
(*Fonctions graphiques*)
  let effaceCorp c fx fy options=
    (*cercle*)
    set_color white;
    fill_circle (int (c.coord_x * options.zoom) ++ options.origine_x)
                (int (c.coord_y * options.zoom) ++ options.origine_y)
                (int (float c.rayon *  options.zoom));
    (*Vecteur Vitesse*)
    moveto (int (c.coord_x * options.zoom) ++ options.origine_x)
           (int (c.coord_y * options.zoom) ++ options.origine_y);
    lineto (int ((c.vitesse_x / options.scale_v + c.coord_x) *
options.zoom) ++ options.origine_x)
           (int ((c.vitesse_y / options.scale_v + c.coord_y) *
options.zoom) ++ options.origine_y);
    (*Vecteur Force*)
    moveto (int (c.coord_x  * options.zoom) ++ options.origine_x)
           (int (c.coord_y  * options.zoom) ++ options.origine_y);
    lineto (int ((fx / options.scale_F + c.coord_x) * options.zoom) ++
options.origine_x)
           (int ((fy / options.scale_F + c.coord_y) * options.zoom) ++
options.origine_y);
  ;;
  let drawCorp c color fx fy options=
    set_color color;
    (*cercle*)
    fill_circle (int (c.coord_x * options.zoom) ++ options.origine_x)
                (int (c.coord_y * options.zoom) ++ options.origine_y)
                (int (float c.rayon * options.zoom));
    (*Vecteur Vitesse*)
    set_color red;
    moveto (int (c.coord_x * options.zoom) ++ options.origine_x)
           (int (c.coord_y * options.zoom) ++ options.origine_y);
    lineto (int ((c.vitesse_x / options.scale_v + c.coord_x) *
options.zoom) ++ options.origine_x)
           (int ((c.vitesse_y / options.scale_v + c.coord_y) *
options.zoom) ++ options.origine_y);
    (*Vecteur Force*)
    set_color c_oie;
    moveto (int (c.coord_x * options.zoom) ++ options.origine_x)
           (int (c.coord_y * options.zoom) ++ options.origine_y);
    lineto (int ((fx / options.scale_F + c.coord_x) * options.zoom) ++
options.origine_x)
           (int ((fy / options.scale_F + c.coord_y) * options.zoom) ++
options.origine_y);
  ;;

(*Fonction plot couple de flottant*)
  let plotc (a,b) (x,y) options =
  if (options.lineto)
  then
    begin
    moveto (int (a * options.zoom) ++ options.origine_x)
           (int (b * options.zoom) ++ options.origine_y);
    lineto (int (x * options.zoom) ++ options.origine_x)
           (int (y * options.zoom) ++ options.origine_y)
    end
  else
    begin
    plot (int (x * options.zoom) ++ options.origine_x)
         (int (y * options.zoom) ++ options.origine_y)
    end;;
```

```
(*Fonction plot liste*)
  let rec plotlist l color options=
  set_color color;
  let rec aux l options =
    match l with
      |[]     -> ()
      |[a]    -> ()
      | a::b::q -> (plotc a b options ; aux q options)
  in aux l options;;


(*Elagage d'une liste (renvoie un élément sur deux)*)
let rec elag_list l =
  match l with
  |[]    -> []
  |[a]   -> [a]
  |a::b::q -> a::(elag_list q);;

let op_list l options =
  try
    match (options.reset, options.elag) with
    |true , _ -> [];
    |_ , true -> (elag_list l);
    |_ ,_     -> l;
  with
  |Out_of_memory -> [];;


(*Diminuer l'ordre de grandeur d'un nombre*)
  let low_number nn max =
    let n = ref nn and exp = ref 0 and coeff = ref 1. in
      while !n > 1e3 do
        n := !n / 10.;
        incr exp;
      done;
      while !n > max do
        n := !n / 1.5;
        coeff := !coeff * 1.5;
      done;
    (!coeff * (10.^^^^(float !exp)));;

let Runge_Kutta_4_ordre_2 f''x x x' (y:float) options=
  let K1 = ref 0. and K2 = ref 0. and K3 = ref 0. and K4 = ref 0.
  and J1 = ref 0. and J2 = ref 0. and J3 = ref 0. and J4 = ref 0. in
(*Calcule de x*)
      J1 := x'* options.dt;
      K1 := f''x(y, x, x')* options.dt;
      J2 := (x' + !K1 / 2.)* options.dt;
      K2 := f''x(y,( x + !J1 / 2.), (x' + !K1 / 2.))* options.dt;
      J3 := (x' + !K2 / 2.)* options.dt;
      K3 := f''x(y, (x + !J2 / 2.), (x' + !K2 / 2.))* options.dt;
      J4 := (x' + !K3)* options.dt;
      K4 := f''x(y, (x + !J3), (x' + !K3))* options.dt;
  (x  + (!J1 + 2. * !J2 + 2. * !J3 + !J4)/ 6.,
   x' + (!K1 + 2. * !K2 + 2. * !K3 + !K4)/ 6.);;
```

```
let bang()=
  let boom = [|"Bang !"; "Boum !"|] in
  set_font "Times";
    for i = 255 downto 0 do
      set_text_size (15 ++ 7 ** (i mod 2) );
      set_color (rgb 255 (i) 0);
      moveto (random__int (size_x() -- 300)) (random__int (size_y()));
      draw_string boom.(i mod 2);
      sound 0 25;
    done
  ;;

(*Fonction solve() - Cas Général : Mode Continu*)
let solve corpsA corpsB dtt =
  open_graph "";
  (*Trajectoire int*int list.*)
  let la = ref [] and lb = ref [] and lm = ref [] and lg = ref [] in
  (*Compteur*)
  let count = ref 0 in
(*Constante de Gravitation Universelle*)
  let Gr = 6.67e-11 in
(*Centre de Masse G*)
  let xG =  (corpsA.masse * corpsA.coord_x + corpsB.masse * corpsB.coord_x)
/ (corpsA.masse + corpsB.masse) in
  let yG =  (corpsA.masse * corpsA.coord_y + corpsB.masse * corpsB.coord_y)
/ (corpsA.masse + corpsB.masse) in
(*Application du TCM*)
  let VxG = (corpsA.masse * corpsA.vitesse_x + corpsB.masse *
corpsB.vitesse_x)/(corpsA.masse + corpsB.masse)
  and VyG = (corpsA.masse * corpsA.vitesse_y + corpsB.masse *
corpsB.vitesse_y)/(corpsA.masse + corpsB.masse) in
  let G = {masse = 0.; rayon = 5; coord_x = xG ; coord_y = yG; vitesse_x =
VxG; vitesse_y = VyG} in
(*Particule Fictive M*)
  let Mu = corpsA.masse * corpsB.masse / (corpsA.masse + corpsB.masse) in
  let Vx = corpsB.vitesse_x - corpsA.vitesse_x + G.vitesse_x in
  let Vy = corpsB.vitesse_y - corpsA.vitesse_y + G.vitesse_y in
  let xM = corpsB.coord_x - corpsA.coord_x + G.coord_x in
  let yM = corpsB.coord_y - corpsA.coord_y + G.coord_y in
  let M = {masse = corpsA.masse*corpsB.masse; rayon = 5; coord_x = xM;
coord_y = yM; vitesse_x = Vx; vitesse_y = Vy} in
(*Equation différentielle en x'' et y'' de M*)
(*En x*)
  let f''x(y,x,v) =
    -(Gr * (M.masse)*(x-G.coord_x)) / ((sqrt(((x-G.coord_x)*(x-G.coord_x) +
(y-G.coord_y)*(y-G.coord_y)))^^^^3.) * Mu ) in
(*En y*)
  let f''y(x,y,v) =
    -(Gr * (M.masse)*(y-G.coord_y)) / ((sqrt(((x-G.coord_x)*(x-G.coord_x) +
(y-G.coord_y)*(y-G.coord_y)))^^^^3.) * Mu ) in
(*Déduction des coordonnées corpsA et corpsB par homothétie*)
(*Corps A*)
  let fxcorpsA = function x -> - (corpsB.masse * (x-G.coord_x)) /
(corpsA.masse + corpsB.masse) in
  let fycorpsA = function y -> - (corpsB.masse * (y-G.coord_y)) /
(corpsA.masse + corpsB.masse) in
(*Corps B*)
  let fxcorpsB = function x -> (corpsA.masse * (x - G.coord_x)) /
(corpsA.masse + corpsB.masse) in
  let fycorpsB = function y -> (corpsA.masse * (y - G.coord_y)) /
(corpsA.masse + corpsB.masse) in
```

```
(*Déduction des vitesses corpsA et corpsB par homothétie*)
(*Corps A*)
  let fvxcorpsA = function x -> - (corpsB.masse * (x-G.vitesse_x)) /
(corpsA.masse + corpsB.masse) in
  let fvycorpsA = function y -> - (corpsB.masse * (y-G.vitesse_y)) /
(corpsA.masse + corpsB.masse) in
(*Corps B*)
  let fvxcorpsB = function x -> (corpsA.masse * (x - G.vitesse_x)) /
(corpsA.masse + corpsB.masse) in
  let fvycorpsB = function y -> (corpsA.masse * (y - G.vitesse_y)) /
(corpsA.masse + corpsB.masse) in
(*Déclaration de "options" et "state"*)
  let options = ref {dt = dtt ; zoom = 1.; origine_x = 0; origine_y = 0;
scale_F = 0.; scale_v = 0.;
  dispTime = false; dispState = false; dispTraj = false ; dispM = false;
dispG = false; blinker = false ;
  dispCorps = true; hypermode = false; lineto = false; elag = false; reset
= false} in
  let state = { corps1 = corpsA; corps2 = corpsB; partFict = M; centreG =
G; F = 0. } in
(*Définitions obsolètes*)
  let yo = M.coord_y and vyo = M.vitesse_y
  and xo = M.coord_x and vxo = M.vitesse_x in
(*End_def*)
  let fx = ref 0. and fy = ref 0. in
(*Calcul de F 1 -> 2 à t0*)
  fx := (abs_float (f''x(yo,xo,vxo)) * Mu);
  fy := (abs_float (f''y(xo,yo,vyo)) * Mu);
(*Evaluation d'une échelle appropriée*)
  (*Force*)
  !options.scale_F <- (low_number (max !fx !fy) ((sqrt ( ((corpsB.coord_x -
corpsA.coord_x)^^^^2.) + ((corpsB.coord_y - corpsA.coord_y)^^^^2.) ))/2.));
  (*Vitesse*)
  !options.scale_v <- (low_number (max vxo vyo) (300.));
(*Runge Kutta 4*)
  let y = ref yo and y' = ref vyo and x = ref xo  and x' = ref vxo  and t =
ref 0. in
  let yy = ref yo and yy' = ref vyo and xx = ref xo  and xx' = ref vxo  in
  (*Pause avant de commencer*)
  wait();
  try
    while true do
      (*Calcule de x*)
      (xx, xx') ~:= (Runge_Kutta_4_ordre_2 f''x !x !x' !y !options);
      (*Calcule de y*)
      (yy, yy') ~:= (Runge_Kutta_4_ordre_2 f''y !y !y' !x !options);
      (*Vitesse et coordonnée de M*)
      (x, x')   ~:= (!xx, !xx');
      (y, y')   ~:= (!yy, !yy');
      (*Traitement des données*)
      (*Effaçage des corps*)
        if not(!options.hypermode)
          then
          begin
            (*"Réels"*)
            effaceCorp corpsB (!fx)   (!fy) !options;
            effaceCorp corpsA (- !fx) (- !fy) !options;
            (*"Fictifs"*)
            if !options.dispM then effaceCorp M (0.) (0.) !options;
            if !options.dispG then effaceCorp G (0.) (0.) !options;
          end;
```

```
        (*Mise à jour des Corps (Fictifs et simulés)*)
        (*Centre de Masse G - coordonnées*)
          G.coord_x <- (VxG * !t + xG);
          G.coord_y <- (VyG * !t + yG);
        (*Corps A - coordonnées*)
          corpsA.coord_x <- (G.coord_x + fxcorpsA !x);
          corpsA.coord_y <- (G.coord_y + fycorpsA !y);
        (*Corps B - coordonnées*)
          corpsB.coord_x <- (G.coord_x + fxcorpsB !x);
          corpsB.coord_y <- (G.coord_y + fycorpsB !y);
        (*Particule fictive - coordonnées*)
          M.coord_x <- !x;
          M.coord_y <- !y;
          if not(!options.hypermode)
          then
          begin
            (*calcul de F 1/2*)
            fx := (f''x(!y,!x,!x') * Mu);
            fy := (f''y(!x,!y,!y') * Mu);
            (*Particule Fictive M Vitesses*)
            M.vitesse_x <- !x';
            M.vitesse_y <- !y';
            (*Corps A - Vitesses*)
            corpsA.vitesse_x <- (G.vitesse_x + fvxcorpsA !x');
            corpsA.vitesse_y <- (G.vitesse_y + fvycorpsA !y');
            (*Corps B - Vitesses *)
            corpsB.vitesse_x <- (G.vitesse_x + fvxcorpsB !x');
            corpsB.vitesse_y <- (G.vitesse_x + fvycorpsB !y');
            (*Actualisation de l'état de la simulation*)
            state.corps1   <- corpsA;
            state.corps2   <- corpsB;
            state.partFict <- M;
            state.centreG  <- G;
            state.F        <- (sqrt((!fx ^^^^ 2.)+(!fy ^^^^2.)));
            (*Affichage des corps*)
            (*"Réels"*)
            drawCorp corpsB bleu (!fx) (!fy) !options;
            drawCorp corpsA marron (- !fx)  (- !fy) !options;
            (*"Fictifs"*)
            if !options.dispM then drawCorp M vert (0.) (0.) !options;
            if !options.dispG then drawCorp G rouge (0.) (0.) !options;
          end;
      (*Enregistrement des trajectoires tous les x calculs*)
        if (!count mod gt1(100. * dtt /(!options.dt)) == 0) then
          begin
          (*Enregistrement*)
          la := (op_list ((corpsA.coord_x,corpsA.coord_y) :: !la)
!options);
          lb := (op_list ((corpsB.coord_x,corpsB.coord_y) :: !lb)
!options);
          lm := (op_list ((M.coord_x,M.coord_y) :: !lm) !options);
          lg := (op_list ((G.coord_x,G.coord_y) :: !lg) !options);
          !options.reset <- false; !options.elag <- false;
          (*Affichage*)
          if (!options.dispTraj && not(!options.hypermode)) then
            begin
            plotlist !la marron !options;
            plotlist !lb bleu !options;
            if !options.dispM then (plotlist !lm vert !options;);
            if !options.dispG then (plotlist !lg rouge !options;);
            end
```

```
        end
      else ();
      incr count;
      (*Vérifications des collisions*)
      if ((sqrt (((corpsB.coord_x - corpsA.coord_x)^^^^2.) +
((corpsB.coord_y - corpsA.coord_y)^^^^2.)))  < float (corpsA.rayon ++
corpsB.rayon))
        then raise collision;
      (*Temps*)
        t  := !t + !options.dt;
      (*Affichage du temps*)
        if (!options.dispTime && not(!options.hypermode)) then draw_time
!t;
        if (!options.dispState&& not(!options.hypermode)) then draw_state
(vectString_of_state state !options);
      (*Interaction utilisateur*)
        options := waitif !options state;
      (*Anti-blink*)
      if (!options.blinker && not(!options.hypermode)) then pause 1;
  done;
    with
  |Exit -> print_string "Fin de la simulation."
  |collision -> bang(); print_string "BANG";;


(*
- Version finale - copie de Système non perturbé 19.ml.
*)
```

Source 7 :

Exemples.ml

```
directory "F:\TIPE\Source\Problème à Deux Corps - Mode Continu";;
include "Système non perturbé.ml";;

(*Exemple 1*)
(*Deux masses sans vitesse initiale en interaction gravitationnelle*)

let A = {masse = 5e12; rayon = 5; coord_x = 250. ; coord_y = 200.;
vitesse_x = 0.; vitesse_y = 0.};;
let B = {masse = 1e13; rayon = 7; coord_x = 50.; coord_y = 300.; vitesse_x
= 0.; vitesse_y = 0.};;

solve A B 0.001;;

(*Exemple 2*)
(*Cas d'une masse négligeable devant la seconde*)

let A = {masse = 1e17; rayon = 25; coord_x = 300. ; coord_y = 200.;
vitesse_x = 0.; vitesse_y = 0.};;
let B = {masse = 300.; rayon = 3; coord_x = 450.; coord_y = 200.; vitesse_x
= 0.; vitesse_y = 200.};;

solve A B 0.0001;;

(*Exemple 3*)
(*Deux masses égales en interaction*)

let A = {masse = 3e18; rayon = 5; coord_x = 150.; coord_y = 200.; vitesse_x
= 0.;  vitesse_y = -500. };;
let B = {masse = 3e18; rayon = 5; coord_x = 400.; coord_y = 200.; vitesse_x
= 0.; vitesse_y = 500.};;

solve A B 0.0001;;

(*Exemple 3 bis*)
(*Deux masses égales en interaction sur une orbite circulaire*)

let A = {masse = 3e18; rayon = 5; coord_x = 100.; coord_y = 200.; vitesse_x
= 0.;  vitesse_y = -500. };;
let B = {masse = 3e18; rayon = 5; coord_x = 500.; coord_y = 200.; vitesse_x
= 0.; vitesse_y = 500.};;

solve A B 0.0001;;

(*Exemple 4*)
(*Centre de masse en Mouvement Rectiligne Uniforme*)

let A = {masse = 1e17; rayon = 25; coord_x = 0. ; coord_y = 0.; vitesse_x =
25.; vitesse_y = 25.};;
let B = {masse = 300.; rayon = 3; coord_x = 150.; coord_y = 0.; vitesse_x =
0.; vitesse_y = 250.};;

solve A B 0.0001;;
```

```
(*Simulation Terre-Lune*)
(*
- Données :
- Masse de la Terre   : ~ 5,9736 x 10 ^ 24 Kg.
- Masse de la Lune    : ~ 7,3477 x 10 ^ 22 Kg.
- Distance Terre-Lune : ~ 384 400 Km.
- Vitesse orbitale de la Lune : 1,022 Km/s.
- Donnée de vérification de la simulation :
- Vitesse de libération : 2,38 km/s
- Période de révolution : 27,321582 Jours (Soit ~ 2 360 584 s).
*)

let Terre = {masse = 5.9736e24; rayon = 6000000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Lune  = {masse = 7.3477e22; rayon = 1737400; coord_x = 384.4e6;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 1.022e3};;

solve Terre Lune 1.;;

(**************************)
(*Première vitesse cosmique*)

(*Echec de lancement*)

let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat   = {masse = 1e3; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 7.900e3};;

solve Terre Sat 0.001;;

(*Mise en orbite correct*)
let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat   = {masse = 1e3; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 7.950e3};;

solve Terre Sat 0.001;;

(*Deuxième vitesse cosmique*)

(*Echec de libération*)

let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat   = {masse = 1e3; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 11.100e3};;

solve Terre Sat 0.001;;

(*Le satellite échappe à l'attraction de la Terre*)
let Terre = {masse = 5.9736e24; rayon = 6378000; coord_x = 0.;
coord_y = 500000.; vitesse_x = 0.;  vitesse_y = 0.};;
let Sat   = {masse = 1e6; rayon = 1; coord_x = 6378100.;  coord_y = 500000.;
vitesse_x = 0.;  vitesse_y = 11.300e3};;

solve Terre Sat 0.001;;
```

Sources de recherches

- Cours de mécanique MPSI.
- Syllabus de l'UNIVERSITE CATHOLIQUE DE LOUVAIN, Faculté des Sciences, Département de Physique : PHY 1111: PHYSIQUE GENERALE 1 MECANIQUE.
- TD ENS Lyon, Méthodes de résolutions numériques.
- http://fr.wikipedia.org
- http://en.wikipedia.org
  Articles :
  - Problème à deux corps (Two-body problem)
  - Méthodes de Résolutions numériques
  - Collision
- http://classiques.uqac.ca/classiques/newton_isaac/principes_math_philo_naturelle/principes_philo_naturelle_t1.html
- http://caml.inria.fr/pub/docs/manual-caml-light/