

Vadim le mathématicien

Deux trois trucs utiles pour notre projet

GARNIER

2022

Sommaire I

1 Présentation

2 Une chute libre

- Une dimension
- Deux dimensions
- Implémentation
- Lois de Newton

3 Des frottements qui piquent

- La notion d'équation différentielle
- Frottements linéaires
- Frottements quadratiques
 - Méthode d'Euler explicite
 - Méthode d'Euler implicite
 - θ -schémas
 - Runge-Kutta

4 Entracte

- Possibles développements futurs
- Kepler

Sommaire II

5 Un moteur 3D ?

6 Mécanique céleste

Que veut-on ? Un programme en C, *OpenGL*, *glut* capable de :

- faire du rendu 2D (voire 3D) de trajectoires préalablement calculées par notre "moteur",
- développer un mini "moteur physique" pour simuler un environnement dans lequel le projectile évoluera,
- dans un premier temps, simuler la trajectoire d'un projectile en fonction de :
 - la masse,
 - l'attraction gravitationnelle,
 - des force de frottements (linéaires, quadratiques)
- complexifier le bordel (pas faire que de la mécanique newtonienne, aller taper dans de la physique et de la chimie qui commencent à être pointues...).

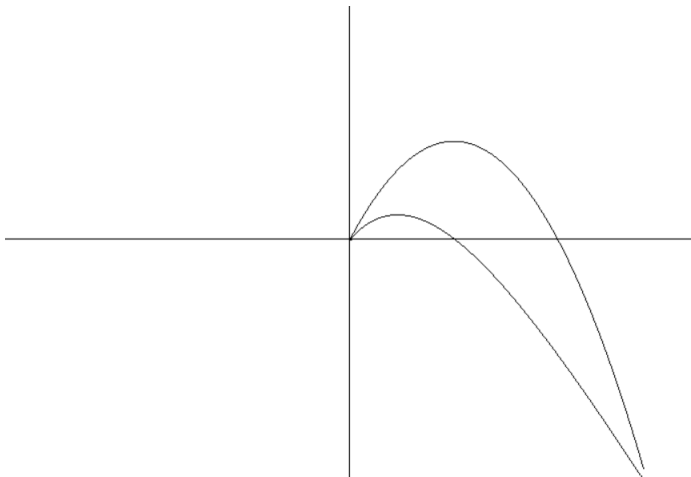
Présentation

Bon, rien que pour ce petit commencement, on a du travail (programmer proprement, implémenter des méthodes type Runge-Kutta, foutre les mains à la pâte en refaisant tous les calculs physique...).

C'est ensuite que ça devrait devenir trop bien (quand on commencera à implémenter des sortes de "maillages" pour approximer la forme d'une surface et que l'on attendra que la trajectoire dépende de la forme de l'objet, des frottements subis...).

Pour plus d'informations, j'ai un autre document qui développe ce que l'on pourrait faire à l'avenir.

Au moment où j'écris, où en est-on ? On a tout, faut juste y mettre de l'ordre, avoir une structure de programme. En soit, on pourrait déjà penser à complexifier le bins, mais non, ça a tellement été programmé avec le cul par moment que faut être plus astucieux et reprendre le travail.



AU TRAVAIL !

Mon job va être de te servir sur un plateau d'argent les mathématiques (et la physique !) au programme du projet ! (tqt, je me permettrai des débordements)

Definition

Un corps en chute libre = le corps n'est soumis qu'à l'effet de la force gravitationnelle (on néglige les frottements etc).

Dans le cas d'un corps en chute libre, l'objet subit une **accélération** vers le sol (dans le référentiel terrestre) d'une valeur de $g \approx -9.8 \text{ m/s}^2$.

Et, juste en sachant ça, on peut se démerder pour trouver l'équation de la trajectoire. Comment ? En se rappelant que la dérivée de la position est la vitesse et que la dérivée de la vitesse est l'accélération. En d'autres termes:

$$\frac{dp}{dt} = v(t), \quad \frac{d^2p}{dt^2} = \frac{dv}{dt} = a(t) \quad (1)$$

où $p(t)$, $v(t)$ et $a(t)$ représentent, respectivement : la position, la vitesse et l'accélération au temps t de ton objet.

Et là : magie des choses. Si tu peux passer de la position à l'accélération (en dérivant successivement deux fois), peux-tu également passer de l'accélération à la position (en faisant successivement *quelque chose* deux fois) ? Oui, tout à fait : ecco, l'intégrale :

$$\int a(t)dt = v(t), \quad \int v(t)dt = p(t) \quad (2)$$

et donc, tout logiquement :

$$\int \int a(t)dt dt = p(t). \quad (3)$$

Clairement, dans notre projet on va se battre les couilles de si il est possible ou non de dériver ou d'intégrer dans la mesure où on va travailler sur des valeurs numériques (la notion de continuité, par exemple, n'existe pas pour un ordinateur (discret par nature)... à moins que tu fasses du calcul symbolique, mais ce n'est pas nous ça (pour l'instant ?)). En théorie, faudrait que la fonction soit suffisamment régulière (continue voire \mathcal{D}^k ou \mathcal{C}^k (pour un certain k)). En revanche, faudra qu'on fasse attention à des comportements limites et, comme toujours, si on a du $\pm\infty$ non désiré, c'est qu'on a foiré quelque chose.

Trajectoire et chute libre

Voyons un cas pratique : déterminons l'équation de la trajectoire d'un corps en chute libre (en ne considérant qu'une seule dimension : x , on fera le cas du plan ensuite).

On sait que l'accélération est de $g \approx -9.80 \text{ m/s}^2$ (et on pourrait le faire pour n'importe quelle autre valeur scalaire, je te laisse imaginer). Y'a plus qu'à dérouler les calculs :

$$a(t) = g \quad (4)$$

$$v(t) = \int_{t_0}^t a(t)dt = g(t - t_0) + v_0 \quad (5)$$

$$p_x(t) = \int_{t_0}^t v(t)dt = \frac{1}{2}g(t - t_0)^2 + v_0(t - t_0) + x_0 \quad (6)$$

avec v_0 la vitesse initiale, x_0 l'altitude initiale et t_0 le temps initial.

En connaissant $p_x(t)$, on peut être en mesure de faire le **graphique de l'évolution de la position de l'objet au cours du temps** (pour peu que l'on connaisse g , v_0 , x_0 et t_0).

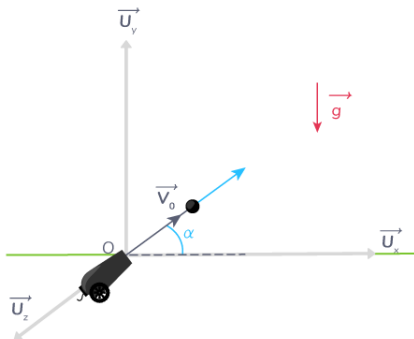
Et, en dimension deux (dans le plan), il y a juste une subtilité à la fin mais ce n'est pas franchement plus compliqué, on va juste avoir "deux fois plus" de variables : celles pour gérer les x et celles pour gérer les y . Ainsi, y'a plus qu'à :

$$a_x(t) = 0, \quad v_x(t) = v_0 \cos(\alpha), \quad p_x(t) = v_0 \cos(\alpha)t \quad (7)$$

$$a_y(t) = g, \quad v_y(t) = gt + v_0 \sin(\alpha), \quad p_y(t) = \frac{1}{2}gt^2 + v_0 \sin(\alpha)t \quad (8)$$

(là, on a pas trop fait gaffe à l'altitude initiale, au temps initial etc, je te le présente juste comme on l'avait fait en Terminale mais il n'y a aucune difficulté pour obtenir des résultats plus généraux comme précédemment)

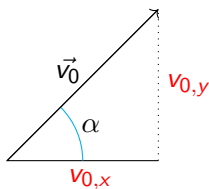
La seule chose qui pourrait te gêner réside dans ces histoires de $\cos(\alpha)$ et $\sin(\alpha)$ (bon, il y a peut-être quelque chose d'autre mais j'éclaircirai le point sur les lois de Newton par la suite). Nous sommes dans la configuration suivante :



Refaisons de la bonne vieille trigonométrie¹ (**CAH SOH TOA**) afin de nous y retrouver.

¹Va falloir revoir pas mal de choses si on veut faire un moteur 3d :).

Tu vas donc avoir ton angle α comme ci-dessous :



J'ai ajouté en rouge des informations importantes. Et, de là, en jouant un petit peu avec la formule $\cos(\alpha) = \frac{\text{côté adjacent}}{\text{hypoténuse}}$, tu obtiens bien la formule escomptée (à savoir, $v_0 \cos(\alpha)$). Une dernière petite question demeure : pourquoi s'intéresser au cosinus ? Rapelle toi simplement de la représentation trigonométrique d'un nombre complexe : le cosinus représente les x et le sinus les y , donc c'est assez logique...

Il ne reste qu'un dernier petit problème à régler et on aura fini cette première étape ! Lorsque tu vas vouloir tracer la trajectoire en fonction du temps, on est bien d'accord que tu vas donc chercher une fonction p trajectoire qui dépend du temps (en gros, chaque point de la trajectoire va être représenté par le couple $(t, p(t))$). Petit problème... Si tu fais bien attention, on a évidemment la variable t mais également une fonction $p_x(t)$ et une fonction $p_y(t)$, comment tracer trois trucs en 2d ? Aïe. Il y a une astuce, on va partir des équations horaires du mouvement (les $p_x(t)$ et $p_y(t)$ afin d'en dériver l'équation de la trajectoire ! Pour ce faire, on va injecter $p_x(t)$ dans $p_y(t)$ en rusant :

nous savons que

$$p_x(t) = v_0 \cos(\alpha)t \quad (9)$$

ce qui revient essentiellement à dire que

$$t = \frac{p_x(t)}{v_0 \cos(\alpha)} \quad (10)$$

de cette manière, on va faire bouffer à $p_y(t)$ les dépendances en t et les remplacer par des $p_x(t)$. Ensuite, il faudra juste être cohérent dans notre code pour que tout fonctionne.

On injecte $p_x(t)$ dans $p_y(t)$:

$$p_y(t) = \frac{1}{2}gt^2 + v_0 \sin(\alpha)t = \frac{1}{2}g \left(\frac{p_x(t)}{v_0 \cos(\alpha)} \right)^2 + v_0 \sin(\alpha) \frac{p_x(t)}{v_0 \cos(\alpha)} \quad (11)$$

Quelques simplifications, et l'on trouve que

$$p_y(t) = \frac{g}{2v_0^2 \cos^2(\alpha)} p_x^2(t) + \tan(\alpha) p_x(t) \quad (12)$$

Cette petite astuce nous permet alors, **pour peu que l'on ait préalablement calculé** $p_x(t)$, de pouvoir tracer la fonction qui a t associée $p_y(t)$ et ainsi d'obtenir un tracé de la trajectoire.

On verra, dans la diapositive suivante, une implémentation concrète du tracé de cette fonction (quelques détails ont été mis de côté par soucis de clarté (d'où les trois petits points au début et à la fin)).

```
...
for (int pas = 0; pas <= precision; pas++) {

    t = (pas * dist_parcours) / precision;
    xpos = init_speed * cos(angle) * t;
    ypos = -g / (2 * init_speed * init_speed * cos(angle) * cos(angle))
           * xpos * xpos + tan(angle) * xpos;

    glVertex2f(t, ypos);
}
...
```

Ça c'était vraiment la partie... échauffement, très simple. C'est essentiellement ce que l'on s'évertue à te faire gober en Terminale S.

On va commencer à entrer dans des choses "moins triviales" (voire carrément difficiles, pour lesquelles l'ordinateur va nous être d'un bien grand secours). Intéressons-nous aux frottements ! On va essentiellement en voir deux types : des frottements proportionnels à la vitesse (frottements linéaires) ou bien des frottements proportionnels au carré de la vitesse (frottements quadratiques). La physique sera toujours au programme mais il va falloir de plus :

- commencer à toucher sa bille en mathématiques (équations différentielles...),
- s'intéresser à des méthodes d'analyse numérique, d'approximation de solutions d'équations différentielles (Runge-Kutta, Euler...).

Mais avant d'aller plus loin, j'aimerais refaire un petit topos sur les lois de Newton. Elles vont pouvoir éclairer ce que l'on a précédemment fait mais elles vont surtout nous servir quasiment immédiatement que se pointent les frottements.

1^{ère} loi – Principe d'inertie

Un corps persiste dans son immobilité ou dans son mouvement de translation rectiligne et uniforme dès lors qu'il n'est soumis à aucune force externe.

2^{ème} loi – Principe fondamental de la dynamique

Si l'on applique des forces externes \vec{F}_i à un corps, sa quantité de mouvement \vec{p} varie dans le sens et dans la direction de la force nette appliquée. Cette variation est d'autant plus élevée que l'intensité de la force appliquée est grande. Mathématiquement, $\vec{p} = m\vec{v}$, et :

$$\sum_{\text{forces externes}} \vec{F}_i = \frac{d\vec{p}}{dt} = \frac{d(m\vec{v})}{dt} \quad (13)$$

3^{ème} loi – Principe des actions réciproques / mutuelles

L'action est toujours égale à la réaction, c'est-à-dire que les actions de deux corps l'un sur l'autre sont toujours égales et de sens contraires.

On va essentiellement utiliser le principe fondamental de la dynamique (le fameux "*pdf*"). Faisons le d'ailleurs tout de suite et voyons en quel sens il peut nous être utile et justifier plus sérieusement certains de nos agissements il y a quelques slides de cela.

Selon le " pfd", la somme des forces externes est égale à la dérivée de la quantité de mouvement, ce qui est également égal à la dérivée de $m\vec{v}$. Par linéarité de l'opérateur de dérivation, on sait que :

$$\frac{d(m\vec{v})}{dt} = m \frac{d\vec{v}}{dt} \quad (14)$$

, or, on connaît la dérivée du vecteur vitesse². On peut alors en conclure que :

$$\sum_{\text{forces externes}} \vec{F}_i = m\vec{a}. \quad (15)$$

Néanmoins, nous sommes dans une chute libre ! Donc, nous savons ce que vaut la somme des forces externes : le vecteur \vec{g} , représentant l'attraction gravitationnelle (et pour pas plus se casser la tête que ça, postulons que la masse vaille 1). De fait, $\vec{g} = \vec{a}$, boum, on intègre on intègre, boum, on a la position. Magie magie.

²Je ne parle aucunement des histoires de dérivations vectorielles etc. On fait comme si tout allait bien.

Des frottements qui piquent³

Deux problèmes vont se poser à nous :

- celui pour lequel les frottements sont linéaires, le "pfd" nous donne :

$$m \frac{d\vec{v}}{dt} = m\vec{g} - k_1 \vec{v} \quad (16)$$

- et, celui pour lequel les frottements sont quadratiques, le "pfd" nous donne :

$$m \frac{d\vec{v}}{dt} = m\vec{g} - k_2 \|\vec{v}\| \vec{v} \quad (17)$$

Il semblerait, plus généralement, qu'il soit possible de complexifier (et d'aller au cas du linéaire et quadratique). Apparemment, on peut représenter la force de frottement tel que :

$$\vec{F}_{\text{frottement}} = -k_{1,\text{frottement}} \|\vec{v}\| \frac{\vec{v}}{\|\vec{v}\|} - k_{2,\text{frottement}} \|\vec{v}\|^2 \frac{\vec{v}}{\|\vec{v}\|} - \dots \quad (18)$$

³Oui, c'est là que ça va commencer à devenir dur.

La notion d'équation différentielle

Commençons par faire un petit détour (oui, déjà !). Nous allons investiguer la notion d'**équation différentielle** afin que tu aies quelques repères (métaphoriquement, que tu connaisses plus que quelques recettes apprises par coeur : que tu sois capable de lire de nouvelles recettes).

Habituellement (au lycée et même plus tard), tu devais résoudre des équations pour lesquelles les solutions étaient des nombres (e.g. $x^2 + 2x - 8 = 0$). Il existe en fait plein d'autres "familles" de solutions envisageables : tu peux restreindre le type de nombre que tu veux (des nombres complexes ? réels ? entiers ?) et surtout tu peux chercher plus que de simples nombres !

Par exemple, tu vas avoir les équations fonctionnelles (e.g. trouver les fonctions f , si elles existent, telles que $f(x^2 + y) - f(y) = 2yf(x) - x^2$). Si en plus tu précises certaines caractéristiques de ta fonction (*j'autorise qu'elle soit trois fois dérivable*, par exemple), la notion d'équation différentielle commence à apparaître.

Qu'est-ce qu'une équation différentielle ? Bah une équation fonctionnelle (= où l'on cherche des fonctions) dans laquelle la fonction et ses dérivées interviennent (e.g. trouver les fonctions f , si elles existent, telles que $xf''(x) + \sin(x)f(x) = 1 + x^2$).

Sans trop faire gaffe, tu sais déjà résoudre des équations différentielles (ou alors, en réfléchissant un peu, tu as tout ce qui faut pour te débrouiller), par exemple, trouver la fonction f vérifiant :

$$f'(x) = f(x), \text{ avec } f(0) = 1 \quad (19)$$

boum, bonjour fameuse fonction exponentielle (elle a d'ailleurs également une équation fonctionnelle toute belle : $f(x + y) = f(x)f(y)$).

Tu remarqueras qu'une chose est apparue : on précise que l'on veut fixer une valeur de f (ça ne paraît pas mais c'est essentiel). Imaginons que tu n'aies pas fixé $f(0) = 1$, alors l'ensemble des fonctions $f : x \mapsto C \cdot \exp(x)$ serait solution. Or, quand on veut utiliser une solution, il est tout de même plus commode de n'avoir qu'une seule solution plutôt qu'un ensemble de solutions paramétré par la constante C .

En d'autres termes, le fait de fixer $f(0) = 1$ fixe également la valeur que peut prendre C (ici, $C = 1$).

Un peu plus tard, tu te rendras compte que de telles considérations sont vraiment essentielles. Essentiellement, tu vas avoir **le(s) théorème(s) de Cauchy-Lipschitz** qui, par exemple dans sa version **linéaire**, énonce l'**existence** et l'**unicité** d'une solution sur tout un intervalle pour une certaine équation différentielle avec une condition initiale du type $f(t_0) = f_0$. (Je passe vite.)

Bref, bref, bref... Fondamentalement et formellement, une équation différentielle se définit alors ainsi :

Definition

Soient N et n deux entiers strictement positifs, soient Ω un ouvert de $\mathbb{R} \times (\mathbb{R}^N)^n$ et $\varphi : \Omega \rightarrow \mathbb{R}^N$. Une **équation différentielle d'ordre n dans \mathbb{R}^N** est définie comme suit :

$$\frac{d^n f}{dt^n} := f^{(n)}(t) = \varphi(t, f(t), f'(t), \dots, f^{(n-1)}(t)). \quad (20)$$

Il s'agit de résoudre ce bordel (pour une fonction φ fixée, bien évidemment). On dira qu'on aura résolu (ou *intégré* l'équation différentielle) lorsque l'on disposera, pour un intervalle sur lequel cela fait sens, d'une fonction f qui satisfasse l'équation différentielle d'ordre n étudiée.

Pour le beau jeu, voici quelques équations différentielles issues de la physique : le mouvement d'une planète autour du Soleil est "régit" par l'équation différentielle suivante :

$$f'' = -\mu \frac{f}{\|f\|^3} \quad (21)$$

, pour étudier des battements de coeur ou des interactions de plaques sur une faille en sismologie :

$$f''(t) + (3f^2(t) - 1)f'(t) + f(t) = 0. \quad (22)$$

Ensuite, on peut même aller plus loin en fabriquant de nouveaux types d'équations différentielles ! Par exemple, lorsque l'on a des fonctions à plusieurs variables, on peut former des **équations aux dérivées partielles** en combinant des fonctions entre elles avec leurs dérivées *par rapport* à la première, la seconde, la troisième (...) variable.

Par commodité, soit une fonction à trois variables $f(x, y, z)$, on ne notera (dans ce cas spécifique) plus $\frac{df}{dy}$ la dérivée de f par rapport à y mais $\frac{\partial f}{\partial y}$, ou encore plus simplement : $\partial_y f$.

La manière dont se diffuse la chaleur dans une barre unidimensionnelle homogène d'une certaine longueur L (et d'autres paramètres) se transcrit alors sous l'équation aux dérivées partielles suivante :

$$\partial_t u - \partial_{xx}^2 u = 0 \quad (23)$$

(avec des conditions initiales). On nomme cette équation : équation de la chaleur, ou encore, équation de Fourier. Déjà que c'est compliqué les équations différentielles... les équations aux dérivées partielles, c'est une autre paire de manche !

Juste pour te donner une idée, une solution de l'équation de la chaleur est la suivante :

$$u(t, x) = \sum_{k=1}^{\infty} b_k e^{-\frac{k^2 \pi^2}{L^2} t} \sin\left(\frac{k\pi}{L} x\right) \quad (24)$$

(pour des coefficients b_k bien précis).

On laisse les équations aux dérivées partielles de côté, il y a déjà suffisamment à faire juste avec des équations différentielles. Mais, pourquoi pas y faire un petit tour plus tard ?! Revenons à nos moutons.

On va se retrouver un petit peu emmerdé car :

- pour la partie sur les frottements linéaires, c'est du bête et méchant, on va apercevoir vite fait le b.a.b.a. du b.a.b.a. et il n'y a pas franchement besoin de plus,
- en revanche, pour la partie avec des frottements quadratiques, tout se complique à tel point qu'il faille donner une approximation de la solution (j'ai cru comprendre qu'on avait un mal fou à construire explicitement une solution).

Frottements linéaires

Faisons le bilan des forces en présence : l'attraction gravitationnelle et des frottements proportionnels à la vitesse. Par application du "pfd", on en déduit la relation suivante :

$$m\vec{g} - k_1\vec{v} = m\vec{a} \quad (25)$$

(je triche un peu et je ferme les yeux sur quelques détails), on se rappelle que la dérivée de la vitesse c'est l'accélération, on en arrive alors à l'équation différentielle suivante (après avoir ré-arrangé un petit peu les termes) :

$$m\frac{dv}{dt} + k_1v(t) = mg. \quad (26)$$

Dans la pratique, concrètement, comment ce genre d'équations différentielles se résolvent ?

Pour des raisons algébriques (tu as des structures d'espaces vectoriels qui traînent dans le bins), tu peux n'étudier que l'équation différentielle suivante $m \frac{dv}{dt} + k_1 v(t) = 0$, appelée **équation homogène associée**. Ensuite, tu trouves une **solution particulière** de l'équation différentielle. L'ensemble des solutions sera alors composée de la combinaison des solutions de l'équation homogène associée et d'une solution particulière. On déroule l'arsenal (que tu trouveras dans tout cours introductif sur la résolution d'équations différentielles linéaires⁴), et l'on trouve :

$$v(t) = v_l \left(1 - \exp \left(- \frac{t}{\tau} \right) \right) \quad (27)$$

avec v_l une certaine vitesse limite et τ le temps nécessaire pour atteindre un certain seuil de vitesse.

⁴Si tu veux, je peux t'expliquer tout ce gloubiboulga, mais ça va me prendre un peu de temps.

Pour trouver la position, il n'y a plus qu'à intégrer $v(t)$! Mais attention ! Pour des considérations physiques (la chute ayant lieu vers le bas), on va devoir intégrer $-v(t)$ et non $v(t)$.

On obtient en fin de compte l'équation de la trajectoire suivante :

$$p_y(t) = -v_I t + v_I \tau \left(1 - \exp \left(-\frac{t}{\tau} \right) \right) + h \quad (28)$$

où h correspond à la condition initiale $p_y(t = 0)$.

Frottements quadratiques

Faisons le bilan des forces en présence : l'attraction gravitationnelle et des frottements proportionnels au carré de la vitesse. Par application du " pfd ", on en déduit la relation :

$$m\vec{g} - k_2\|v\|\vec{v} = m\vec{a}. \quad (29)$$

De la même manière que dans le cas des frottements linéaires, on en déduit alors l'équation différentielle suivante :

$$m\frac{dv}{dt} + k_2v^2 = mg. \quad (30)$$

Contrairement à précédemment, l'équation différentielle n'est pas linéaire et... quand ce n'est pas linéaire, c'est une autre paire de manche ! Je ne sais pas s'il existe une solution analytique (\approx solution exacte)... pour ne pas se casser la tête, on va la résoudre numériquement : on délègue tous les calculs à la machine, on approxime la solution.

Dans mon livre préféré sur les équations différentielles (cf. la photo), tu as toute une partie sur l'étude numérique d'équations différentielles.

- méthode d'Euler explicite,
- méthodes numériques à un pas,
- schéma d'Euler implicite,
- méthodes de Runge-Kutta...

Et tout ça sans parler de tous les problèmes de convergences qui se posent. Il ne faut pas oublier que toutes les méthodes ne se valent pas en fonction des situations, il faut pouvoir adapter le choix d'une méthode au problème rencontré.

Alors, quelle méthode choisir ? Selon F. Berthelin, voici le topos :

Méthode	Avantage	Défaut
Euler explicite	facile et rapide à mettre en oeuvre	pas de discrétisation doit être très petit pour bonne approximation
Euler implicite	n'a pas le défaut de la méthode d'Euler explicite	coûteux en calcul (équations non linéaires)
RK4	précis, facile et rapide	même défaut qu'Euler explicite (précision accrue), problème quand les données sont "bruitées"

On va s'intéresser et implémenter chaque méthode. Mais avant cela, présentons la teneur du problème que l'on veut résoudre (= quel problème d'approximation nous fait face).

Admettons que la détermination de la trajectoire se ramène à la résolution de l'équation différentielle suivante (= déterminer la fonction f solution de l'équation différentielle) :

$$f' = \varphi(t, f), \text{ avec la condition } f(t_0) = f_0 \quad (31)$$

(on met un peu sous le tapis les questions de régularité). Le but de la suite de la section va être d'expliquer comment on peut se débrouiller numériquement pour construire une fonction $\omega_{\text{approx}}(t_i)$ approximant $f(t_i)$.

Par exemple, dans le cas qui va nous intéresser en premier lieu (frottements quadratiques), le problème s'écrit ainsi :

$$v'(t) = \varphi(t, v(t)) = g - \frac{k_2}{m}v^2(t) \quad (32)$$

Juste pour te montrer à quel point passer par un schéma numérique équivaut à se simplifier la vie (= ne pas toucher à de gros objets mathématiques) : si je ne dis pas de bêtises, il semble exister une solution exacte au problème lié aux frottements quadratiques. Sauf que... la solution utilise des fonctions spéciales (et leur dérivées) (très belles mais un peu rebutantes aux premiers abords) :

$$\text{Ai}(z) := \frac{1}{3^{2/3}\Gamma(\frac{2}{3})} {}_0F_1\left(;\frac{2}{3};\frac{z^3}{9}\right) - \frac{z}{\sqrt[3]{3}\Gamma(\frac{1}{3})} {}_0F_1\left(;\frac{4}{3};\frac{z^3}{9}\right) \quad (33)$$

$$\text{Bi}(z) := \frac{1}{\sqrt[6]{3}\Gamma(\frac{2}{3})} {}_0F_1\left(;\frac{2}{3};\frac{z^3}{9}\right) + \frac{\sqrt[6]{3}z}{\Gamma(\frac{1}{3})} {}_0F_1\left(;\frac{4}{3};\frac{z^3}{9}\right) \quad (34)$$

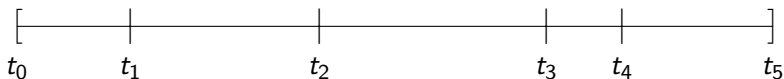
avec $\Gamma(t)$ la fonction gamma d'Euler et ${}_0F_1(; b; z)$ une fonction hypergéométrique généralisée (un sacré morceau !).

OK NON, j'ai dit de la merde. C'est encore plus simple, j'avais foutu un t en trop (c'est un problème de tangente hyperbolique surtout, pas franchement plus (?)). C'est quand on sera en deux dimensions que ce sera une autre histoire.

Méthode d'Euler explicite

On recherche notre fameuse fonction f et l'on sait qu'elle satisfait l'équation (31) ! Mais, ne savons nous pas d'autres choses ?! La réponse va dépendre de ce que l'on s'autorise comme caractéristiques portant sur f (par exemple, si on s'attend à ce que la solution de f soit super biscornue, parte dans tous les sens, fasse des sauts, varie brusquement... *a priori*, elle doit être plus "complexe" qu'une fonction bien belle bien régulière). Si on la prend suffisamment "régulière", on sait qu'elle va pouvoir s'écrire comme une somme de polynôme (= elle va pouvoir s'exprimer comme une série de Taylor). Or, les polynômes c'est cool, très naturel et relativement simple à manier, donc bingo ! Sauf que problème, on a peut-être un peu trop "approximer le bins". Ne connaîtrait-on pas un autre procédé qui revienne essentiellement au-même ? Oh mais, l'équation de la tangente, ne serait-ce pas ce que l'on appelle une approximation *au premier ordre* de notre fonction ? Prenons un peu du champ, et regardons ce que l'on appelle les **développements limités** (tous les objets que j'ai cité dans ce paragraphe sont relativement liés).

Les approximations que l'on va faire, on ne va pas pouvoir les faire n'importe où ! On va s'intéresser à un intervalle, disons $[t_0; t_5]$. Lorsque l'on va chercher à faire faire des calculs à l'ordinateur, on se rend rapidement compte qu'il a du mal avec tout ce qui est continu, donc il va falloir discrétiser ! Ainsi, on va découper notre intervalle (ici, par exemple en 5 bouts différents et on va se donner 4 valeurs : $f(t_1), f(t_2), f(t_3)$ et $f(t_4)$).



Tout notre travail va se ramener en le fait de donner une bonne approximation des valeurs $f(t_1), f(t_2), f(t_3)$ et $f(t_4)$ en cherchant une manière simple de passer d'une valeur $f(t_k)$ à une valeur $f(t_{k+1})$.

Et c'est là que la notion de développement limité de la solution f va s'avérer utile ! En effet, un tel outil nous permet de trouver le fameux chaînon manquant :

$$f(t_{n+1}) = f(t_n + h_n) \quad (35)$$

$$= f(t_n) + h_n f'(t_n) + o(h_n) \quad (36)$$

$$= f(t_n) + h_n \varphi(t_n, f(t_n)) + o(h_n) \quad (37)$$

avec $o(h_n)$ une fonction tendant vers 0 et h_n l'écart entre deux points de la discrétisation.

La **méthode d'Euler explicite** se définit alors comme ci-dessous :

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + h_n \varphi(t_n, \omega_n^{\text{approx}}) \text{ pour } n = 0, 1, \dots, N \quad (38)$$

où N correspond au nombre de points de la discrétisation. (Montrer qu'une telle méthode est convergente n'est pas de tout repos : Berthelin a besoin de trois pages de calculs et pas mal de lemmes et d'analyse réelle.)

Voyons ce que cela donne en pratique sur un exemple très simple : on cherche à résoudre l'équation différentielle $f'(t) = \lambda f(t)$. On construit le schéma d'Euler explicite associé. Néanmoins, on va supposer le pas constant, c'est à dire que h_n est constant (= ne dépend pas de n) (et donc qu'il y a toujours le même écart entre deux points). De fait :

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + h\varphi(t_n, \omega_n^{\text{approx}}) \quad (39)$$

avec $\varphi(t, f) = \lambda f(t)$. Ainsi :

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + h\lambda\omega_n^{\text{approx}} \quad (40)$$

$$= (1 + h\lambda)\omega_n^{\text{approx}} \quad (41)$$

Désormais, ce n'est plus qu'un problème sur les **suites** !!! On en déduit alors que $\omega = (1 + h\lambda)^N$, or $h = \frac{\text{longueur de l'intervalle } [t_0; t_n]}{\text{nombre de points de la discrétisation}} = \frac{T}{N}$. (On fera bien attention à ne pas confondre n et N .)

En introduisant la nouvelle relation portant sur h , on en déduit la valeur de ω :

$$\omega = \left(1 + \frac{\lambda T}{N}\right)^N \quad (42)$$

MAGIE ! C'est la définition de l'exponentielle quand on fait tendre N vers l'infini. Ainsi, **la solution de l'équation différentielle de départ est $\exp(\lambda T)$!**

Ok, j'ai un peu triché en ne justifiant pas certaines choses et en essayant un peu de te berner avec les notations... Autrement, ça aurait fait quelque chose de vraiment très lourd que je ne peux pas me permettre dans un *beamer* introductif.

On peut un complexifier un petit peu l'approche en introduisant le **schéma d'Euler implicite** défini comme suit :

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + h_n \varphi(t_{n+1}, \omega_{n+1}^{\text{approx}}) \text{ pour } n = 0, 1, \dots, N \quad (43)$$

Essentiellement, ensuite, on déroule de la même manière que pour la méthode d'Euler explicite...

Comme on l'a vu, les schémas d'Euler explicite et implicite ont chacun leur défaut et leur qualité ! Et si l'on pouvait tirer "le meilleur d'entre eux" en ayant un schéma **mélangeant une partie d'implicite** (= le calcul de $\omega_{n+1}^{\text{approx}}$ n'est pas direct dans la mesure où le second membre du schéma dépend également de $\omega_{n+1}^{\text{approx}}$) et **une partie d'explicite** ? Réponse : bonjour chers θ -schémas. On définit ces derniers ainsi :

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + h_n \left[\theta \varphi(t_{n+1}, \omega_{n+1}^{\text{approx}}) + (1 - \theta) \varphi(t_n, \omega_n^{\text{approx}}) \right] \quad (44)$$

- Si $\theta = 0$, alors on retrouve la méthode d'Euler explicite,
- si $\theta = 1$, alors on retrouve la méthode d'Euler implicite,
- si $\theta = 1/2$, alors on trouve le *schéma de Crank-Nicholson*,
- sinon, amusons-nous !

Désormais, on change un peu l'approche ! Nous allons nous mettre à utiliser des points intermédiaire afin d'obtenir des solutions plus précises et convergeant plus rapidement. On change également notre manière d'attaquer le problème en cherchant une **formulation intégrale** et non différentielle.

Le problème de base $f' = \varphi(t, f)$ peut se réécrire ainsi :

$$f(t_{n+1}) - f(t_n) = \int_{t_n}^{t_{n+1}} \varphi(s, f(s)) ds. \quad (45)$$

On va chercher à approximer l'intégrale apparaissant au RHS (= *right hand side*, à droite du signe égal). Ensuite, le problème se ramènera de nouveau au fait de trouver un lien entre $f(t_{n+1})$ et $f(t_n)$, et on déroule comme des grands !

Je te passe les détails (uniquement techniques). Beauté de l'histoire, on réussit alors à déterminer l'expression suivante permettant d'estimer la valeur en des points intermédiaires à ceux de la discrétisation :

$$f(t_{n,l}) \approx f(t_n) + h_n \sum a_{lk} \varphi(t_n + c_k h_n, f(t_n + c_k h_n)) \quad (46)$$

pour des bornes de sommation bien définies et des coefficients a_{lk} , c_k bien précis.

C'est à partir de maintenant que ça devient vraiment pointilleux, je suis obligé de passer beaucoup de choses sous silence. Pour l'instant, l'idée n'est que de savoir que "ça existe !". Les détails viendront ensuite.

On peut alors définir le **schéma numérique de Runge-Kutta** :

$$\omega_{n,l}^{\text{approx}} = \omega_n^{\text{approx}} + h_n \sum a_{lk} \varphi(t_n + c_k h_n, \omega_{n,k}^{\text{approx}}) \quad (47)$$

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + h_n \sum b_l \varphi(t_n + c_l h_n, \omega_{n,l}^{\text{approx}}) \quad (48)$$

pour $n = 0, 1, \dots, N$.

En fait, on va pouvoir construire une infinité de schémas de Runge-Kutta. Le plus connu, **RK4**, est implémenté dans l'une des diapositives suivantes. On va ajouter quatre étapes intermédiaires, ce qui évidemment complexifie le calcul mais accroît nettement sa précision (on a par exemple aisément des résultats précis à 10^{-3} ou 10^{-4} près).

Si l'on détermine toutes les constantes a_l, k, b_l et c_k , le schéma **RK4** s'énonce comme suit : on pose ω_0^{approx} égal à la valeur initiale $f(t_0)$. Ensuite, on a :

$$\omega_{n+1}^{\text{approx}} = \omega_n^{\text{approx}} + \frac{h_n}{6} (k_1 + 2k_2 + 2k_3 + k_4) \quad (49)$$

avec :

$$k_1 = \varphi(t_n, \omega_n^{\text{approx}}) \quad (50)$$

$$k_2 = \varphi\left(t_n + \frac{h_n}{2}, \omega_n^{\text{approx}} + \frac{k_1}{2}\right) \quad (51)$$

$$k_3 = \varphi\left(t_n + \frac{h_n}{2}, \omega_n^{\text{approx}} + \frac{k_2}{2}\right) \quad (52)$$

$$k_4 = \varphi(t_n + h_n, \omega_n^{\text{approx}} + k_3) \quad (53)$$

on trouve alors que $\omega_i^{\text{approx}} \approx f(t_i)$.

```

float* solve(float init_timeY, float (*func)(float, float), float time_step_size,
             float init_time, float end_time) {
    float time = init_time;
    float *w = malloc(sizeof(float) * roundf((end_time - init_time) /
                                             time_step_size) + 1);

    w[0] = init_timeY;
    float k1, k2, k3, k4, ti;
    unsigned int step = 0;
    while (time < end_time) {
        ti = init_time + step * time_step_size;

        k1 = time_step_size * func(ti, w[step]);
        k2 = time_step_size * func(ti + time_step_size / 2, w[step] +
                                   k1 / 2);
        k3 = time_step_size * func(ti + time_step_size / 2, w[step] +
                                   k2 / 2);
        k4 = time_step_size * func(ti + time_step_size, w[step] + k3);

        w[step + 1] = w[step] + (k1 + 2 * k2 + 2 * k3 + k4) / 6;

        time += time_step_size;
        step += 1;
    }
    return w;
}

```

Ce que l'on a fait jusqu'à présent, même si ça peut sembler relativement "technique" et "difficile", ça reste des mathématiques et de la physique pour les bébés. Il faut se dire que l'on s'est intéressé, jusqu'alors, qu'à un seul type de mouvement, que l'on a fait que de la mécanique newtonienne, que l'on est resté sur des systèmes très (très) simples et que l'on a pas (vraiment) pris en compte l'environnement (e.g. variation de pression, température...).

Je suis fermement convaincu que l'on peut aller (assez) loin ! Dans tous les cas, le champ des possibles est immense !!! Sans même penser aux processus aléatoires (trajectoires de mouvement brownien⁵ et cie), il y a une putain de pléiade de choses à faire : dynamique cosmologique, dynamique des fluides, équation d'une particule dans un certain système, équation de Boltzmann...

⁵Ça pourrait être bien de faire tout un module "stochastic".

Je t'avoue que la phrase de ton père disant que c'est basique ce que l'on fait, même si c'est clairement vrai, ça me donne une envie folle de pousser à bout nos capacités.

Prenons un peu le temps de nous demander dans quelle direction nous pourrions nous orienter⁶ ! Car, mine de rien, on a déjà fait un petit paquet de chemin (pour des gens qui ne font que commencer). Honnêtement, j'ai envie que mon ordinateur crache du sang et qu'il calcule comme un monstre. Moralité : il a du travail et nous en avons encore plus !

Ne pas oublier que, même si le but est de toucher un peu à tout (en développant plein de modules différents), faut être capable de les finir !!!

⁶ANATOMIE COMPUTATIONNELLE, c'est mon bébé ce truc.

Possibles développements futurs


On a trois gros sujets :

- statique,
- dynamique, et,
- cinématique.

Chacun de ces sujets semble pouvoir être exprimé d'une certaine manière. Jusqu'alors nous n'avons fait que de la **mécanique newtonienne**, c'est une première approximation qu'il serait intéressant d'envisager de dépasser. On pourrait aller voir du côté de la **mécanique lagrangienne** et de la **mécanique hamiltonienne**. Ça ferait déjà un sacré bonhomme de chemin.

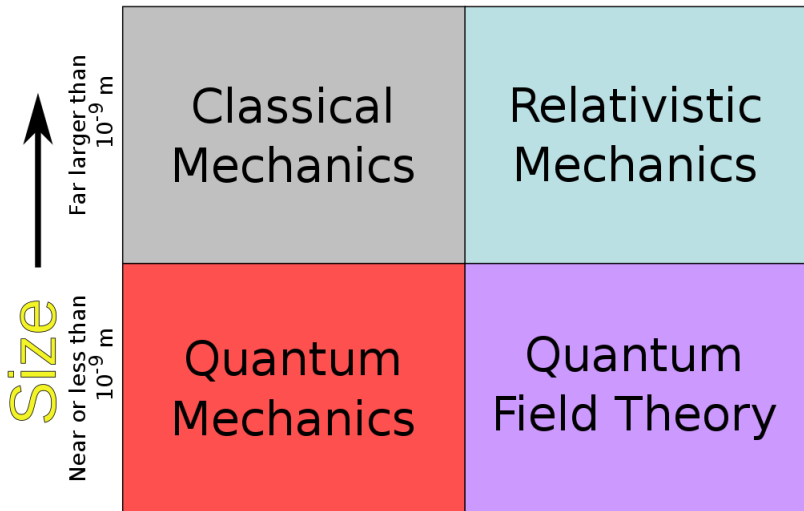
Est-ce que l'on ira voir du côté des **mécaniques relativistes**, de la **mécanique quantique**, de la **théorie quantique des champs** ? Ce n'est pas le sujet pour le moment, on a déjà un nombre incroyable de choses à faire rien qu'en restant sur de la **mécanique classique** !

Speed



Far less than 3×10^8 m/s

Comparable to 3×10^8 m/s



On pourrait aller faire pas mal de choses :

- thermodynamique,
- mécanique céleste,
- astrodynamique,
- mécanique du solide,
- mécanique des fluides,
- systèmes dynamiques,
- dynamique moléculaire,
- ...

On sait qu'il faut rester réaliste et que l'on ne peut pas s'éparpiller à l'infini dans tous les sens... Il faudrait réussir à déterminer une première orientation.

Et si on partait vers des choses comme ça (je n'en dis pas trop, peut-être que tu sais d'où ça sort, nous verrons en temps voulu) :

$$m_j \ddot{\vec{q}}_j = -G \sum_{k \in \{1, \dots, N\} \setminus \{j\}} \frac{m_j m_k (\vec{q}_j - \vec{q}_k)}{\|\vec{q}_j - \vec{q}_k\|^3} \quad (54)$$

$$U_g = \frac{\mu_e}{r} \left(1 + \sum_{n=2}^{\infty} \left(\left(\frac{r_{eq}}{r} \right)^n (-J_n P_n(\sin \delta)) + \right. \right. \quad (55)$$

$$\left. \left. \sum_{m=1}^n (C_{mn} \cos(m\lambda) + S_{mn} \sin(m\lambda)) P_{nm}(\sin \delta) \right) \right) \quad (56)$$

$$\mathcal{H} = \frac{\|\mathbf{v}\|^2}{2} - \frac{\mathcal{G} m_0}{\|\mathbf{r}\|} - \mathcal{G} \sum_{j=1}^N m_j \left(\frac{1}{\|\Delta_j\|} - \frac{\mathbf{r} \cdot \mathbf{s}_j}{\|\mathbf{s}_j\|^3} \right) \quad (57)$$

On ne va pas tout de suite parler des objets présent sur la diapositive précédente, on va un peu prendre notre temps et nous concentrer sur des choses relevant (relativement) du programme de Terminale (mais c'est dans une proche logique de la mécanique céleste) : **l'astronomie, les lois de Kepler et quelques trajectoires.**

Commençons par énoncer les trois lois de Kepler !

1^{ère} loi

Les trajectoires des planètes du système solaire sont des ellipses dont le Soleil est l'un des foyers.

2^{ème} loi

Le segment qui relie le Soleil à une planète couvre des aires égales pendant des intervalles de temps égaux.

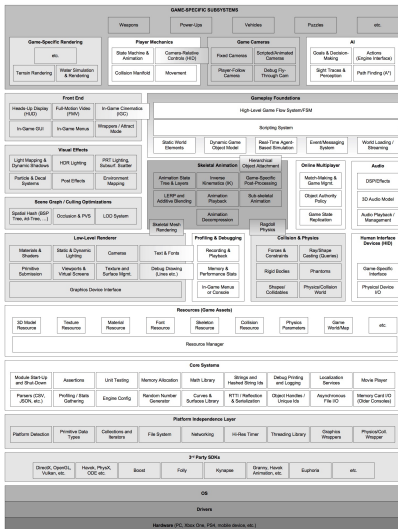
3^{ème} loi

Les carrés des périodes de rotation sont proportionnels aux cubes des demi-grands axes des ellipses.

Bonne (ou pas) nouvelle, la géométrie à laquelle tu es habitué ne va pas suffire ! Il nous faut aller découvrir de nouveaux objets : **les ellipses** (*penser à en faire un module dans le module de géométrie*).

Un moteur 3D

C'est à partir de ce moment que l'on entre dans un autre monde et que notre projet prend une nouvelle ampleur.



À la louche, de quoi aurions-nous besoin ? De modules :

- de **physique** (newtonienne, képlérienne...),
- d'**algèbre linéaire** (voire un peu plus) et de **géométrie 2d et 3d** (*meshes*, triangulation...) pour gérer tout l'aspect mathématique de l'affichage et de la physique dont on aura ensuite besoin,
- de **calcul parallèle** (?),
- d'**analyse numérique** (approximations en tout genre),
- de **fichiers de configurations** (pour lire des fichiers dans lesquels est encodée la géométrie d'un environnement, par exemple) et **de sauvegardes**,
- d'**algorithmes de générations**, de **mesh adaptation** (...),
- d'**interfaçage homme / machine** (gestions caméras, clavier pour se déplacer dans l'espace ou autre...),
- de **gestion de textures et des surfaces** avec **gestion de la lumière, d'ombres** (*shaders...*) (e.g. pour les textures d'une simulation de galaxie).

N'ayant pas la moindre idée de comment faire concrètement un tel moteur, on va procéder en deux étapes :

- continuer à déterminer (de plus en plus précisément) nos attentes afin de savoir dans quelle direction nous orienter (typiquement, même si ça paraît tentant de faire une sorte de moteur de jeux, ce n'est pas forcément le plus adapté à notre cas), et,
- se documenter (il y a notamment le livre *Game Engine Architecture* de Jason Gregory qui semble pas mal, apparemment, c'est le livre " *de-facto standard in the subject of game engines* ").

J'ai regardé pas mal de projets GitHub et franchement ça a l'air carrément envisageable (il y a juste le problème des proportions... ce sont des projets juste... énormes... en revanche, sur la technique et la complexité on devrait totalement faire l'affaire, au moins pour les premières parties).

Je me dis qu'avec des idées connes ne serait pas une si mauvaise idée... je t'explique ! Si par exemple on fait de la mécanique céleste ou si on touche à des choses très très petites (de l'ordre de 10^{-33} par exemple), l'ordinateur ne va rien y piger. Et si on incluait une gestion symbolique des nombres dans notre moteur, ok ça risque d'alourdir pas mal le bordel mais ça ouvre la voie à de la programmation symbolique et go go go. (Je me demande à quel point c'est envisageable mais en gros on ferait de l'arithmétique en utilisant des **char[]** (à moins que *stdint* et des trucs pour les *double* fassent l'affaire, dans tous les cas faudra penser aux problèmes de **précision**).)

Commençons avec le code minimal suivant (cf. page suivante) :

```
#include <GL/glut.h>
#include "render.h" // C'est ici que va commencer le début du travail

void renderScene(void) {

    // Écrire une fonction pour initialiser la scène
    glutSwapBuffers();
}

int main(int argc, char **argv) {

    // Paramètres d'initialisation (taille de la fenêtre)
    // à mettre dans le render.h
    const unsigned short sizeX = 720;
    const unsigned short sizeY = 480;

    // Écrire une fonction pour initialiser le rendu, comme celle suivante :
    initRender(argc, argv, sizeX, sizeY);

    glutDisplayFunc(renderScene);
    glutMainLoop();

    return 0;
}
```

Je vais essayer de te montrer comment construire un **moteur 3d**, pas à pas, en te présentant (dans un ordre logique) divers bouts de code (on va carrément laisser de côté la gestion (très) bas niveau⁷, on veut juste un truc qui fonctionne sur Windows, Linux (et pourquoi pas Mac), *a priori*, on ne se destine pas à déployer le moteur sur de l'embarqué).

Mais avant d'aller plus loin, regardons un peu comment des projets comme **Mesa3D** (implémentation open source de OpenGL, OpenGL ES, Vulkan, OpenCL (et bien plus encore)) s'en sort ! Pour ce faire, dans les diapositives suivantes, vont être présentés quelques fichiers : *Pixel, Point, Line, Polygon*. (On ne va voir que quelques bouts, chaque fichier fait plusieurs centaines de lignes de code.)

HS : j'ai passé un peu de temps à regarder leur code source et j'ai l'impression que, par exemple pour tout l'algèbre linéaire qui peut être utile (matrices, rotation...), ils définissent d'abord un truc purement mathématique (en se battant les couilles du contexte et ensuite ils vont réutiliser les fonctions définies pour les adapter à un contexte bien particulier : e.g. pour des polygones dans une situation bien particulière).

⁷Aussi bien du point de vue hardware que pour de la programmation assembleur (le moteur de jeu Quake en utilise par exemple).

PIXEL

```
...
void _mesa_init_pixel ( struct gl_context *ctx ) {
    /* Pixel group */
    ctx->Pixel.RedBias = 0.0;
    ctx->Pixel.RedScale = 1.0;
    ctx->Pixel.GreenBias = 0.0;
    ctx->Pixel.GreenScale = 1.0;
    ctx->Pixel.BlueBias = 0.0;
    ctx->Pixel.BlueScale = 1.0;
    ctx->Pixel.AlphaBias = 0.0;
    ctx->Pixel.AlphaScale = 1.0;
    ctx->Pixel.DepthBias = 0.0;
    ctx->Pixel.DepthScale = 1.0;
    ctx->Pixel.IndexOffset = 0;
    ctx->Pixel.IndexShift = 0;
    ctx->Pixel.ZoomX = 1.0;
    ctx->Pixel.ZoomY = 1.0;
    ctx->Pixel.MapColorFlag = GL_FALSE;
    ctx->Pixel.MapStencilFlag = GL_FALSE;
    init_pixelmap(&ctx->PixelMaps.StoS);
    init_pixelmap(&ctx->PixelMaps.ItoI);
}
```

```
init_pixelmap(&ctx->PixelMaps.ItoR);
init_pixelmap(&ctx->PixelMaps.ItoG);
init_pixelmap(&ctx->PixelMaps.ItoB);
init_pixelmap(&ctx->PixelMaps.ItoA);
init_pixelmap(&ctx->PixelMaps.RtoR);
init_pixelmap(&ctx->PixelMaps.GtoG);
init_pixelmap(&ctx->PixelMaps.BtoB);
init_pixelmap(&ctx->PixelMaps.AtoA);

if (ctx->Visual.doubleBufferMode) {
    ctx->Pixel.ReadBuffer = GL_BACK;
}
else {
    ctx->Pixel.ReadBuffer = GL_FRONT;
}

/* Miscellaneous */
ctx->_ImageTransferState = 0;
}
```

POINT

```
...  
void _mesa_init_point(struct gl_context *ctx) {  
    ctx->Point.SmoothFlag = GL_FALSE;  
    ctx->Point.Size = 1.0;  
    ctx->Point.Params[0] = 1.0;  
    ctx->Point.Params[1] = 0.0;  
    ctx->Point.Params[2] = 0.0;  
    ctx->Point._Attenuated = GL_FALSE;  
    ctx->Point.MinSize = 0.0;  
    ctx->Point.MaxSize = MAX2(ctx->Const.MaxPointSize, ctx->Const.MaxPointSizeAA);  
    ctx->Point.Threshold = 1.0;
```

/ Page 403 (page 423 of the PDF) of the OpenGL 3.0 spec says:*

```
*  
*     "Non-sprite points (section 3.4) - Enable/Disable targets  
*     POINT_SMOOTH and POINT_SPRITE, and all associated state. Point  
*     rasterization is always performed as though POINT_SPRITE were  
*     enabled."  
*  
*     In a core context, the state will default to true, and the setters and  
*     getters are disabled.
```



```
    */  
ctx->Point.PointSprite = (ctx->API == API_OPENGL_CORE ||  
                          ctx->API == API_OPENGLS2);  
  
ctx->Point.SpriteOrigin = GL_UPPER_LEFT; /* GL_ARB_point_sprite */  
ctx->Point.CoordReplace = 0; /* GL_ARB_point_sprite */  
}
```

LINE

```
...  
void _mesa_init_line( struct gl_context * ctx ) {  
    ctx->Line.SmoothFlag = GL_FALSE;  
    ctx->Line.StippleFlag = GL_FALSE;  
    ctx->Line.Width = 1.0;  
    ctx->Line.StipplePattern = 0xffff;  
    ctx->Line.StippleFactor = 1;  
}
```

POLYGON

```
...
void _mesa_init_polygon( struct gl_context * ctx ) {
    /* Polygon group */
    ctx->Polygon.CullFlag = GL_FALSE;
    ctx->Polygon.CullFaceMode = GL_BACK;
    ctx->Polygon.FrontFace = GL_CCW;
    ctx->Polygon.FrontMode = GL_FILL;
    ctx->Polygon.BackMode = GL_FILL;
    ctx->Polygon.SmoothFlag = GL_FALSE;
    ctx->Polygon.StippleFlag = GL_FALSE;
    ctx->Polygon.OffsetFactor = 0.0F;
    ctx->Polygon.OffsetUnits = 0.0F;
    ctx->Polygon.OffsetClamp = 0.0F;
    ctx->Polygon.OffsetPoint = GL_FALSE;
    ctx->Polygon.OffsetLine = GL_FALSE;
    ctx->Polygon.OffsetFill = GL_FALSE;

    /* Polygon Stipple group */
    memset( ctx->PolygonStipple, 0xff, 32*sizeof(GLuint) );
}

```

C'est extrêmement intéressant d'aller voir les fichiers *draw* et *drawpix*. Par exemple, dans ce dernier, tu as une fonction `_mesa_DrawPixels`. La logique est la suivante : tu récupères la valeur du contexte (`GET_CURRENT_CONTEXT(ctx)`) dans lequel tu vas travailler (afin d'orienter la suite des hostilités), s'en suit une liste longue comme le bras de cas de debug, d'erreurs probables à anticiper / gérer, ensuite, tu vas pouvoir mettre à jour la valeur de ton pixel (cf. la diapositive suivante pour voir comment le bordel se fait), puis de nouveau une nouvelle liste (encore plus longue) de cas à gérer.

Le gros du gros qu'il nous reste à comprendre réside dans l'**affichage** (je suis conscient que OpenGL s'occupe de tout cela pour nous, mais ça ne nous empêche pas de nous "faire un début d'intuition" en voyant comment "font les grands").

```
void _mesa_update_pixel( struct gl_context *ctx )
{
    GLuint mask = 0;

    if (ctx->Pixel.RedScale    != 1.0F || ctx->Pixel.RedBias    != 0.0F ||
        ctx->Pixel.GreenScale  != 1.0F || ctx->Pixel.GreenBias  != 0.0F ||
        ctx->Pixel.BlueScale   != 1.0F || ctx->Pixel.BlueBias   != 0.0F ||
        ctx->Pixel.AlphaScale  != 1.0F || ctx->Pixel.AlphaBias  != 0.0F)
        mask |= IMAGE_SCALE_BIAS_BIT;

    if (ctx->Pixel.IndexShift || ctx->Pixel.IndexOffset)
        mask |= IMAGE_SHIFT_OFFSET_BIT;

    if (ctx->Pixel.MapColorFlag)
        mask |= IMAGE_MAP_COLOR_BIT;

    ctx->_ImageTransferState = mask;
}
```

```
void GLAPIENTRY _mesa_DrawElements(GLenum mode, GLsizei count, GLenum type,
                                   const GLvoid * indices) {
    GET_CURRENT_CONTEXT(ctx);
    FLUSH_FOR_DRAW(ctx);

    _mesa_set_draw_vao(ctx, ctx->Array.VAO,
                      ctx->VertexProgram._VPModeInputFilter);

    if (ctx->NewState)
        _mesa_update_state(ctx);

    if (!_mesa_is_no_error_enabled(ctx) &&
        !_mesa_validate_DrawElements(ctx, mode, count, type))
        return;

    _mesa_validated_drawrangeelements(ctx, mode, false, 0, ~0,
                                      count, type, indices, 0, 1, 0);
}
```

Comme j'ai essayé de t'en parler (et on en voit un exemple juste au-dessus avec le test `if(!_mesa_is_no_error_enabled(ctx)...)`), la notion de **gestion d'erreur**, d'essayer d'avoir une longueur d'avance sur de probables **comportements indésirables du programme** va être super importante⁸ ! Il va également falloir que l'on ait notre **système de gestion de la mémoire** et **des ressources**. Certes, on ne fait pas une application dite *critique* (de l'embarqué, performances limitées, environnement hostile ou autre), mais autant partir sur de bonnes bases plutôt que de devoir tout refaire des cons dans plusieurs semaines ou mois (en plus ça va nous faire apprendre plein de choses).

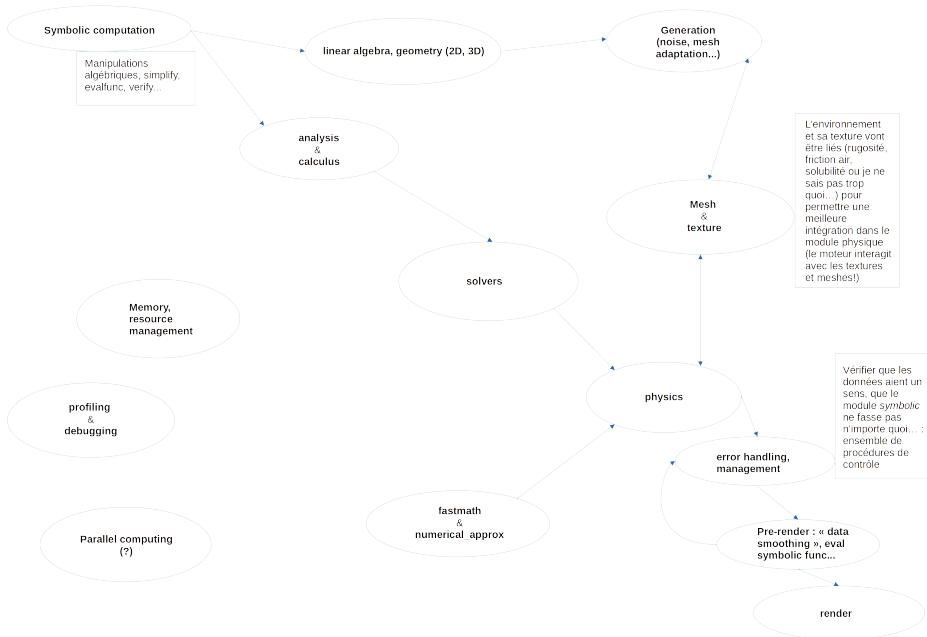
Faudra aussi inclure des systèmes de **profiling** et de **debugging**.

Et, je me demandais, ne serait-ce pas intéressant d'avoir un **langage de scripting interne au moteur** ? Par exemple, sans avoir à tout recompiler, on aurait juste à re-compiler / ré-interpréter le script à l'intérieur du moteur et, hop, la simulation évolue !

⁸Elle est d'ailleurs présente dans le gros schéma page 56 dans la partie *Core systems* : c'est la case **Assertions**.

Ça me démange de **faire un moteur 3D dont toutes les logiques (internes) seraient symboliques** ! Ce ne serait qu'au moment de l'affichage que l'information serait "downgradée" vers quelque chose d'habituel (par exemple, avant toute la phase de pré-rendu / rendu (c'est-à-dire lorsque les calculs / la simulation se fait), l'ordinateur ne "réfléchit" que symboliquement (par exemple en manipulant des $\sqrt{2}$ et non des 1.4142...)) et ce n'est que seulement ensuite, au moment du rendu, que tout est converti en des types de données habituels que *OpenGL* + *glut* vont pouvoir rendre). Bon, en revanche, un tel système serait sacrément lourd et demandeur en ressource (mais vu que l'on ne fait pas de temps réel, on s'en fiche relativement beaucoup).

J'ai fait un premier diagramme pour voir comment les choses pourraient se présenter (cf. page suivante).



Voici comment, schématiquement, ça devra(it) se présenter !

```
#include "..."

int main(int argc, char **argv) {

    // initialisation du programme et de l'affichage
    physics_env environment = /* définition de l'environnement physique */;
    symbol_expr pb = /* forme mathématique du problème */;

    link(environment, pb);
    apply(environment, /* appliquer des contraintes, forces... */);
    setObject(environment, /* définition de l'objet d'étude */);

    simulate(environment); // en se basant sur l'environnement et l'objet
                           // en question, déterminer l'équation de la
                           // trajectoire (et faire le rendu)

    return 0;
}
```

Pour voir ce que ça peut donner, je vais faire un premier jet en Python en utilisant **sympy** !

```

from sympy import Function, dsolve, solve, Eq, Derivative, integrate, symbols, tanh, cosh, cos, log, sqrt, I, pi, Symbol
from sympy.plotting import plot
import numpy as np
import time

start_time = time.time()

t = Symbol('t', real=True)
g = Symbol('g', real=True)
coeff = Symbol('k', real=True)

#m = 1 # passer ça en Symbol('m', real=True)

v = Function('v')
#eq = Eq(Derivative(v(t), t), - g - coeff * v(t) * v(t))
#sol_sympy = dsolve(eq, v(t))
#print(sol_sympy)
#print()

#Cl = Symbol('Cl')
#Cl_ic = solve(sol_sympy.rhs.subs({t: 0}), Cl)[0]
#sol_sympy = sol_sympy.subs({Cl: Cl_ic})
#print(sol_sympy)

#res_sympy = sqrt(g)/(sqrt(coeff)*tanh(sqrt(g)*sqrt(coeff)*(t - I*pi/(2*sqrt(g)*sqrt(coeff)))) # obtenu grâce au code juste au-dessus
#res_sympy = res_sympy.subs(g, 9.806) # Étonnement, ça déconne si les valeurs g, m et coeff ne sont pas entières
#res_sympy = res_sympy.subs(coeff, 0.47)

#res_sympy_int = integrate(res_sympy, t)
#print(res_sympy_int)

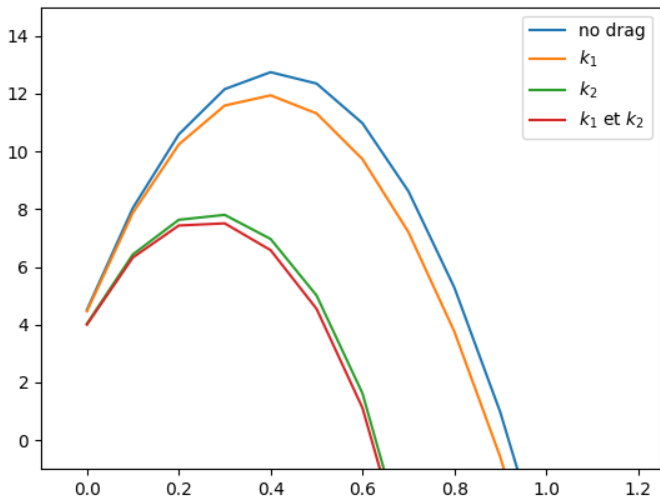
#wolffy_solution = ((g * m)**(1/2) * tanh(((g * coeff)**(1/2) * m + (g * coeff)**(1/2) * t)/(m**(1/2)))) / (coeff**(1/2))
#wolffy_pos_test = integrate(wolffy_solution, t) # marche pas trop
#wolffy_pos_wolffy_int = (g**(3/2) * coeff**(1/2) * log(cosh((t + 1) * (g * coeff)**(1/2)))) / (g * coeff)**(3/2)
#wolffy_pos_wolffy_int = wolffy_pos_wolffy_int.subs(g, 9.806)
#wolffy_pos_wolffy_int = wolffy_pos_wolffy_int.subs(coeff, 0.47)

test = (g**(3/2) * sqrt(coeff) * log(cos(sqrt(g * coeff) * (1 + t)))/(g * coeff)**(3/2)
test = test.subs(g, -9.806)
test = test.subs(coeff, 0.47)

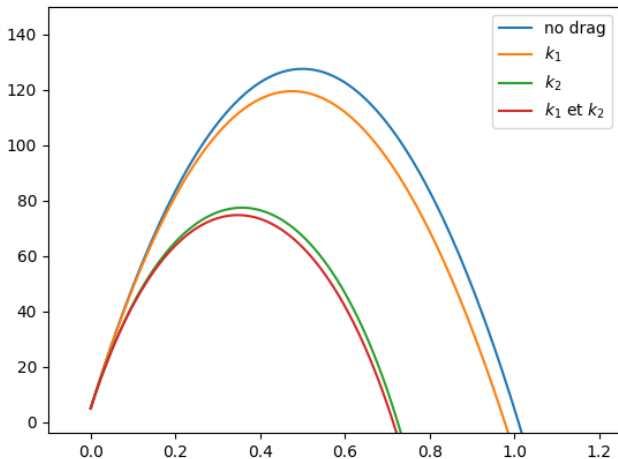
print("--- %s secondes ---" % (time.time() - start_time))
#plot(res_sympy_int, (t, 0.01, 10), line_color="red")
plot(test, (t, 0.01, 10), line_color="red")
plot(wolffy_pos_wolffy_int, (t, 0, 3), line_color="blue")

```

Au vu de tout ce qui est commenté, n'aurais-je pas fait n'importe quoi ?
Je crois que si...

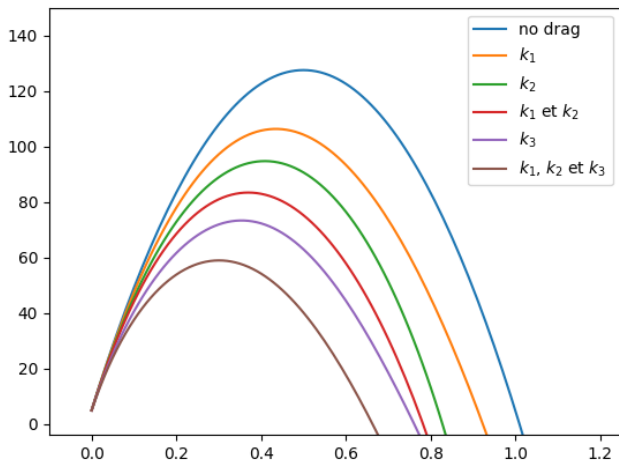


Bon, en reprenant un peu les choses, il semblerait que ça marche un poil mieux !



Si l'on demande un peu plus de détail, on note que les valeurs se comportent bizarrement (regarde les ordonnées par exemple : en divisant le pas d'un facteur 10, on multiplie les ordonnées d'un facteur 10).

Et une dernière pour la route !



La prochaine étape consiste en le fait de réussir à tracer ces mêmes fonctions mais... en utilisant le moteur symbolique de **sympy**. Pour l'instant (sans **sympy**), on a ça :

```
import matplotlib.pyplot as plt
from scipy import integrate
from math import exp
import numpy as np
import time

start_time = time.time()
plt.xlim(-0.1, 1.25)
plt.ylim(-4, 150)

initFunc = 5
g = -9.806
drag = 0.6
drag2 = 0.3
drag3 = 0.2

# Fonctions sur lesquelles va être appliqué le schéma RK4
#  $y' = \text{fun}(t, y)$ 
def funNoDrag(t, y):
    return g
```

```

def funDrag1(t, y):
    return g - drag * y

def funDrag2(t, y):
    return g - drag2 * y * y

def funDrag_combined12(t, y):
    return g - drag * y - drag2 * y * y

def funDrag3(t, y):
    return g - drag3 * y * y * y

def funDrag_combined123(t, y):
    return g - drag * y - drag2 * y * y - drag3 * y * y * y

# Schéma de Runge Kutta (RK4)
def rk4(init_func, func, init_time, end_time, time_step_size):

    time = init_time
    sol = [init_func]

    rk_1, rk_2, rk_3, rk_4 = 0, 0, 0, 0
    ti = 0
    step = 0

```



```

while time < end_time:

    ti = init_time + step * time_step_size
    temp_ti = time_step_size / 2

    rk_1 = time_step_size * func(ti, sol[step])
    rk_2 = time_step_size * func(ti + temp_ti, sol[step] + rk_1 / 2)
    rk_3 = time_step_size * func(ti + temp_ti, sol[step] + rk_2 / 2)
    rk_4 = time_step_size * func(ti + time_step_size, sol[step] + rk_3)

    sol.append(sol[step] + (rk_1 + 2 * rk_2 + 2 * rk_3 + rk_4) / 6)

    time += time_step_size
    step += 1

return sol

def plotPos(ajust, init_func, func, init_time, end_time, time_step_size,
            label_fun=""):
    t = np.arange(init_time, end_time + ajust * time_step_size, time_step_size)
    y = rk4(init_func, func, init_time, end_time, time_step_size)

    plt.plot(t, integrate.cumtrapz(y), label=label_fun)

init_time = 0

```

```
end_time = 3
time_step_size = 0.01

plotPos(1, initFunc, funNoDrag, init_time, end_time, time_step_size,
        label_fun="no drag")
plotPos(1, initFunc, funDrag1, init_time, end_time, time_step_size,
        label_fun="$k_1$")
plotPos(1, initFunc, funDrag2, init_time, end_time, time_step_size,
        label_fun="$k_2$")
plotPos(1, initFunc, funDrag_combined12, init_time, end_time, time_step_size,
        label_fun="$k_1$ et $k_2$")
plotPos(1, initFunc, funDrag3, init_time, end_time, time_step_size,
        label_fun="$k_3$")
plotPos(1, initFunc, funDrag_combined123, init_time, end_time, time_step_size,
        label_fun="$k_1$, $k_2$ et $k_3$")

print("--- %s secondes ---" % (time.time() - start_time))
plt.legend(loc="upper right")
plt.show()
```

Ça m'a l'air (méga) long (pour le peu que c'est) : environ trois secondes de calculs.

