UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Travail de Bachelor

2016

Simulateur d'un pendule inversé

Auteure
Alina Ana-Maria Petrescu
alina.petrescu@unifr.ch

Sous la direction de
Prof. Ulrich Ultes-Nitsche

Semestre d'automne 2016

Département d'Informatique • Université de Fribourg - Universität Freiburg • Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland

http://unifr.ch/diuf/

Table des matières

1 Introduction **1**
1.1 Motivations 1
1.2 Overview 1
1.3 Organisation du document 3

2 Étude du pendule inversé **5**
2.1 Étude mécanique du pendule inversé 5
2.2 Résolution numérique du système d'EDO 9
2.3 Le système dynamique associé au pendule inversé 11
2.3.1 Modèle linéaire d'état 11
2.3.2 Réglage en boucle fermée par commande proportionnelle 14
2.3.3 Considérations qualitatives sur la stabilisation du pendule inversé 19

3 Simulateur du pendule inversé **23**
3.1 Structure du code 23
3.2 Interface graphique utilisateur 27
3.3 Implémentation de la résolution numérique du système d'EDO 28

4 Conclusions **33**
4.1 Synthèse 33
4.2 Éléments informatiques utilisés 33
4.3 Points forts 34
4.4 Développements futurs 34
4.5 Remerciements 35

A Documentation de l'implémentation (API) du projet **37**

B Paramètres modifiables **79**

Table des figures

1.1 Schéma d'un pendule 2

2.1 Système mécanique 5
2.2 Système mécanique - Forces et accélérations 6
2.3 Chariot isolé - Forces et accélérations 7
2.4 Pendule isolé - Forces et accélérations 8
2.5 Étude analytique : le mouvement durant une période 18

3.1 Schéma UML du projet 24
3.2 Emboîtement des containers 27

Chapitre 1

Introduction

1.1 Motivations

L'objectif de ce projet a été de réaliser un simulateur d'un pendule inversé à l'aide du langage de programmation Java. Son élaboration est partie de zéro avec l'étude mécanique du problème réel qui a conduit à un modèle physique approprié dont les équations différentielles ont été résolues numériquement. Une attention particulière a été accordée à la conception d'une interface graphique utilisateur ergonomique et conviviale. De plus, la versatilité et l'extensibilité de la solution retenue ont été des aspects prioritaires de l'approche proposée. Ce travail m'a permis d'utiliser une grande partie des notions théoriques qui m'ont été présentées tout au long de mon cursus du Bachelor en Informatique, et d'approfondir, plus particulièrement, des aspects pratiques liés à la conception et à l'implémentation d'une application Java complexe. De plus, j'ai dû apprendre certaines notions de base d'automatique.

1.2 Overview

Quoi que l'étude d'un pendule soit un chapitre standard de la mécanique classique, ce sujet peut couvrir un grand nombre de problèmes apparentés, certains d'entre eux assez élémentaires, mais d'autres extrêmement complexes.

Considérons un corps de masse m fixé à l'extrémité d'une tige rigide de longueur l qui est liée à un socle par l'intermédiaire d'une articulation de type liaison rotule. Ainsi, le corps m peut effecteur un mouvement de rotation dans un plan vertical autour de cette articulation. On peut simplement modéliser ce système mécanique par un pendule caractérisé par la masse m

et la longueur l. De plus, on note θ l'angle formé, à un moment donné, entre la tige et la verticale (voir Figure 1.1).

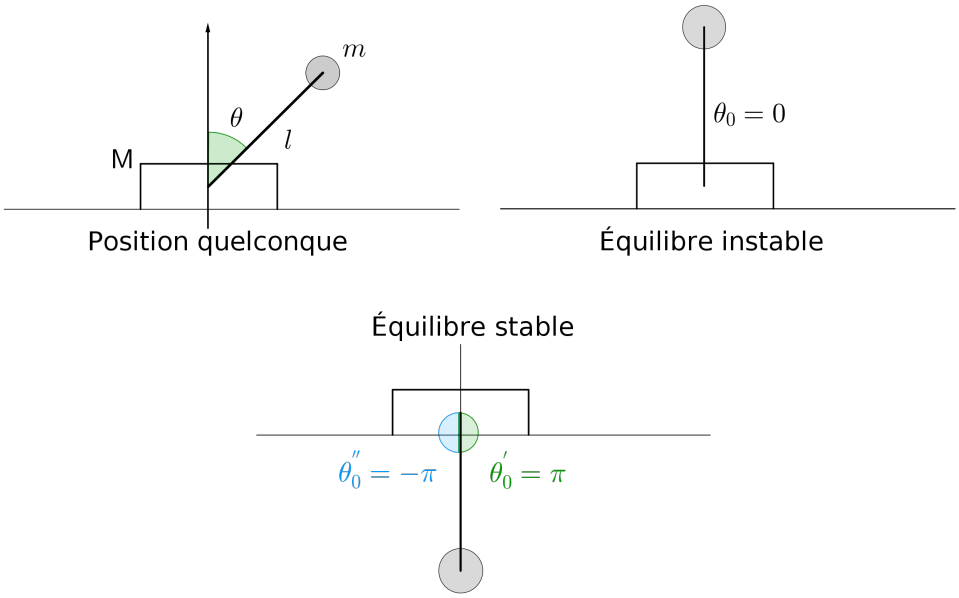


FIGURE 1.1 – Schéma d'un pendule

Afin de déterminer les éventuelles positions d'équilibre d'un tel pendule, il suffit d'exprimer son énergie potentielle E_{pot} sous la forme :

$$E_{pot}(\theta) = m \cdot g \cdot l \cdot \cos(\theta) \tag{1.1}$$

où g est l'accélération gravitationnelle.

En fait, les éventuelles positions d'équilibre correspondent aux angles θ pour lesquelles l'énergie potentielle atteint ses valeurs extrêmes. Par conséquent, on cherche les angles qui annulent la première dérivée par rapport à θ de l'énergie potentielle :

$$\frac{dE_{pot}}{d\theta} = -m \cdot g \cdot l \cdot \sin(\theta) = 0 \tag{1.2}$$

et on garde les solutions $\theta_0 = 0$ et $\theta_0^{'} = \pi$.

On calcule maintenant la deuxième dérivée par rapport à θ de l'énergie potentielle :

$$\frac{d^2E_{pot}}{d\theta^2} = -m \cdot g \cdot l \cdot \cos(\theta) \tag{1.3}$$

ainsi que ses valeurs pour les angles θ_0 et $\theta_0^{'}$:

$$\left.\frac{d^2E_{pot}}{d\theta^2}\right|_{\theta=\theta_0} = \left.\frac{d^2E_{pot}}{d\theta^2}\right|_{\theta=0} = -m \cdot g \cdot l < 0 \tag{1.4}$$

$$\left.\frac{d^2E_{pot}}{d\theta^2}\right|_{\theta=\theta_0^{'}} = \left.\frac{d^2E_{pot}}{d\theta^2}\right|_{\theta=\pi} = m \cdot g \cdot l > 0 \tag{1.5}$$

Vu que dans la relation 1.4 la deuxième dérivée est négative, la valeur de l'énergie potentielle est maximale et on peut conclure que la position correspondant à l'angle $\theta_0 = 0$ est une position d'équilibre *instable*. De même, vu que dans la relation 1.5 la deuxième dérivée est positive, la valeur de l'énergie potentielle est minimale et on peut conclure que la position correspondant à l'angle $\theta_0^{'} = \pi$ est une position d'équilibre *stable*. Il convient de remarquer que ces conclusions sont en concordance avec notre intuition et sont parfaitement vérifiées par l'expérience.

On parle du **PENDULE INVERSÉ** pour faire référence à un pendule qui bouge autour de sa position d'équilibre instable et le but de mon rapport est justement la réalisation d'un simulateur du mouvement d'un tel pendule.

L'aspect délicat de l'étude d'un pendule inversé concerne la stabilisation de celui-ci au voisinage de sa position d'équilibre instable. Pour cela, on accepte que le socle auquel la tige du pendule est rattaché par l'intermédiaire de l'articulation rotule n'est pas fixe. Ainsi, en fonction du mouvement imprimé au socle, on trouve dans la littérature spécialisée (voir, par exemple, [1]) deux techniques de stabilisation principales : la stabilisation *par contrôle* (voir, par exemple, [2] ou [3]) et la stabilisation *par oscillation du support* (expliquée pour la première fois par P. Kapitza [4]). Dans mon travail, j'utilise la première technique de stabilisation en considérant le socle comme un chariot qui peut être accéléré ou décéléré convenablement par l'intermédiaire d'une force de commande "bien" choisie (voir le chapitre 2.3.3).

1.3 Organisation du document

L'étude mécanique du mouvement du pendule inversé est réalisée dans le chapitre 2 qui contient aussi des détails concernant la résolution numérique du système d'équations différentielles ordinaires (EDO) obtenu suite à l'étude mécanique, ainsi que des considérations concernant le système dynamique associé.

La réalisation du simulateur d'un pendule inversé (à l'aide du langage de programmation Java) est présentée dans le chapitre 3 qui explique la structure du code, en mettant l'accent sur l'interface graphique proposée et

sur l'implémentation de la résolution numérique du système d'EDO déduit au chapitre précédent.

Les diverses conclusions, technologies utilisées et améliorations futures sont présentées dans le chapitre 4.

De plus, l'annexe A représente la documentation (API) de l'implémentation du projet générée par *Doxygen* (voir la référence [5]) et l'annexe B passe en revue les principaux paramètres modifiables du simulateur.

Chapitre 2

Étude du pendule inversé

Afin de pouvoir réaliser un simulateur du pendule inversé, on fait d'abord l'étude mécanique de son mouvement. On obtient ainsi le système d'équations différentielles ordinaires qui décrivent le modèle choisi. Par la suite, la résolution de ce système est abordée d'un point de vue numérique.

2.1 Étude mécanique du pendule inversé

On considère le système mécanique présenté dans la FIGURE 2.1 et formé de trois éléments :

- un corps de masse m ;
- une tige rigide de masse négligeable et de longueur l ;
- un chariot de masse M.

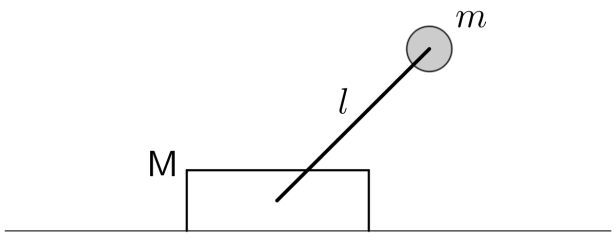


FIGURE 2.1 – Système mécanique

Plus précisément, les dimensions du chariot et du corps peuvent être négligées et ces deux éléments seront donc considérés comme des points matériels. En fait, le corps de masse m est le pendule inversé et il est soudé à une extrémité de la tige. L'autre extrémité de la tige est reliée au chariot

par l'intermédiaire d'une articulation (une liaison rotule) qui lui permet un mouvement de rotation dans un plan vertical.

Si le chariot se déplace (par rapport à un système de référence cartésien Oxy) au long de l'axe horizontal Ox avec une vitesse $\dot{x}$ et une accélération $\ddot{x}$, le pendule m effectue un mouvement composé obtenu par la superposition d'un déplacement horizontal (avec ces mêmes vitesse et accélération) et un mouvement circulaire (non uniforme) au long d'un cercle de rayon l et dont le centre est l'articulation qui relie la tige au chariot. On note θ l'angle formé entre la verticale orientée positivement vers le haut (l'axe Oy) et la direction de la tige orientée positivement du chariot vers le pendule, angle mesuré positivement dans le sens des aiguilles d'une montre. L'angle θ varie dans le temps et $\dot{\theta}$ et $\ddot{\theta}$ sont la vitesse angulaire et, respectivement, l'accélération angulaire du mouvement circulaire.

Par la suite, on va utiliser le principe de D'ALEMBERT qui permet d'étendre le champ d'application des lois de Newton à des systèmes de référence non galiléens. En fait, si on rajoute les forces d'inertie, l'étude de la dynamique d'un système peut être ramené à l'étude d'un problème de statique. Dans la FIGURE 2.2, on a représenté en rouge gras les accélérations et en rouge normal les forces d'inertie correspondantes.

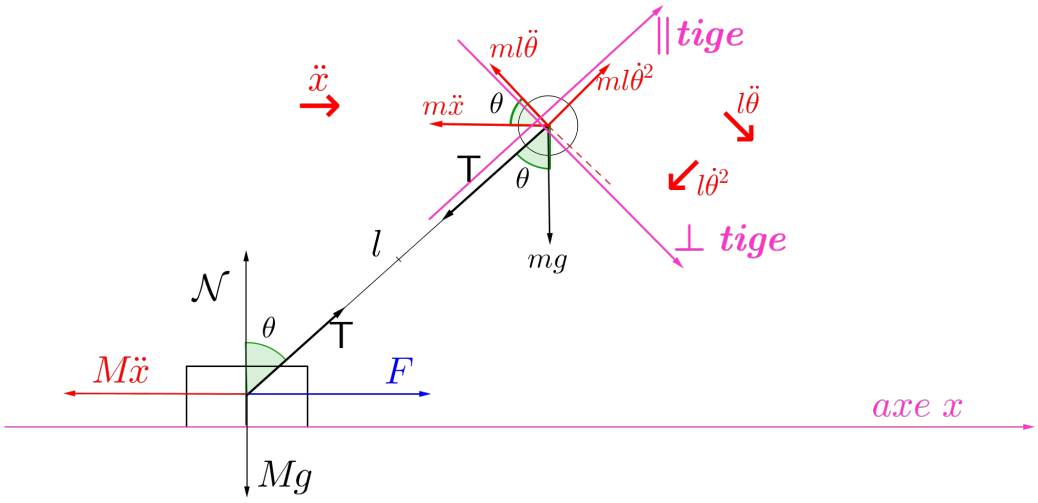


FIGURE 2.2 – Système mécanique - Forces et accélérations

Afin d'obtenir les équations du mouvement du pendule, on isole d'abord le chariot en remplaçant la tige par la force de tension de norme T correspondante et le sol par la force de réaction normale N. En négligeant les frottements (à la fois dans l'articulation ainsi que celui entre le chariot et le sol), on a représenté dans la FIGURE 2.3 les forces agissant sur le chariot, y

compris le poids Mg, la force d'inertie $M\ddot{x}$ et la force de traction F. Cette dernière force est très importante pour notre étude car c'est elle qui doit stabiliser le pendule inversé et son calcul sera présenté plus tard. De plus, dans la figure mentionnée, la vitesse $\dot{x}$ précise le sens du mouvement du chariot et l'accélération $\ddot{x}$ permet de connaître le sens de la force d'inertie (qui est opposée à l'accélération).

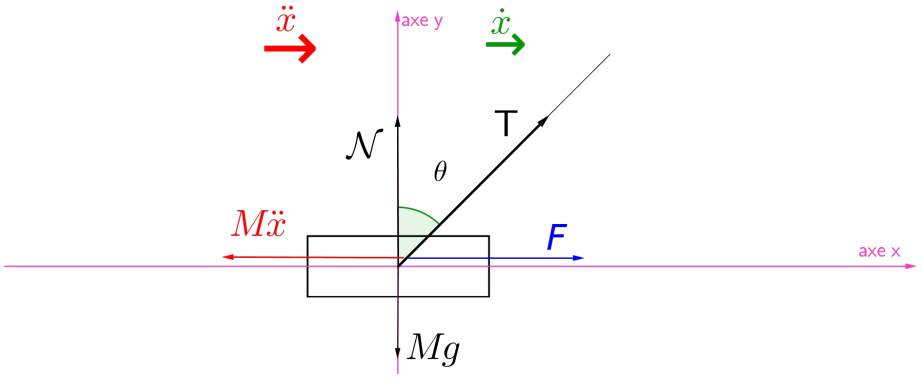


FIGURE 2.3 – Chariot isolé - Forces et accélérations

Conformément au principe de D'ALEMBERT, la somme vectorielle de toutes les forces agissant sur le chariot doit être nulle :

$$\overrightarrow{F} + M\overrightarrow{g} + \overrightarrow{T} + \overrightarrow{N} + M\overrightarrow{\ddot{x}} = 0 \tag{2.1}$$

En choisissant un repère cartésien avec l'axe horizontal Ox orienté positivement vers la droite et l'axe vertical Oy orienté positivement vers le haut, on projette l'équation vectorielle 2.1 sur les deux axes et on obtient un système de deux équations algébriques.

Selon l'axe horizontal :

$$F + T \cdot \sin(\theta) - M \cdot \ddot{x} = 0 \tag{2.2}$$

Selon l'axe vertical :

$$N + T \cdot \cos(\theta) - M \cdot g = 0 \tag{2.3}$$

Vu que le frottement est négligé, il suffit d'utiliser par la suite seulement l'équation 2.2.

En procédant de manière analogue, on isole le pendule de masse m, en remplaçant la tige par la force de tension correspondante de norme T (et qui respecte le principe de l'action et de la réaction). À part cette tension et le

poids mg, il faut prendre en compte aussi les forces d'inertie dues à l'accélération horizontale $\ddot{x}$ ainsi qu'aux accélérations tangentielle $l\ddot{\theta}$ et normale $l\dot{\theta}^2$ du mouvement circulaire. Toutes ces forces et accélérations sont indiquées dans la FIGURE 2.4 (où, comme auparavant, les forces d'inertie sont représentées en rouge et les accélérations en rouge et gras).

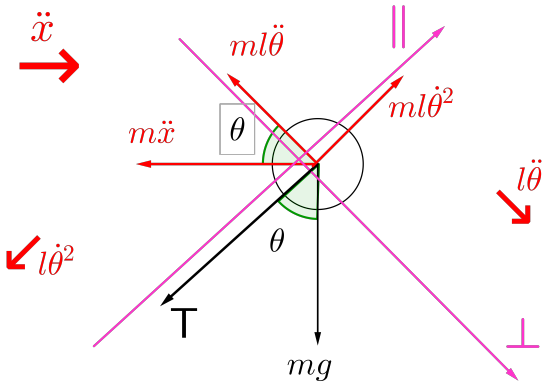


FIGURE 2.4 – Pendule isolé - Forces et accélérations

À nouveau, conformément au principe de D'ALEMBERT, la somme vectorielle de toutes les forces agissant sur le pendule doit être nulle. On choisit un repère cartésien (mobile) avec le premier axe perpendiculaire sur la tige et avec le sens positif correspondant à une rotation dans le sens des aiguilles d'une montre, et l'autre axe au long de la tige et orienté positivement du chariot vers le pendule. On projette l'équation vectorielle qu'on vient de mentionner sur les deux axes et on obtient un système de deux équations algébriques.

Selon l'axe perpendiculaire à la tige :

$$m \cdot g \cdot \sin(\theta) - m \cdot l \cdot \ddot{\theta} - m \cdot \ddot{x} \cdot \cos(\theta) = 0 \tag{2.4}$$

Selon l'axe parallèle à la tige :

$$m \cdot l \cdot \dot{\theta}^2 - T - m \cdot g \cdot \cos(\theta) - m \cdot \ddot{x} \cdot \sin(\theta) = 0 \tag{2.5}$$

À partir de l'équation 2.5, on obtient :

$$T = m \cdot l \cdot \dot{\theta}^2 - m \cdot g \cdot \cos(\theta) - m \cdot \ddot{x} \cdot \sin(\theta) \tag{2.6}$$

En remplaçant cette dernière expression dans la relation 2.2 et en mettant en évidence l'accélération $\ddot{x}$, on obtient :

$$\ddot{x} = \frac{m\, l\, \dot{\theta}^2\, \sin(\theta) - m\, g\, \sin(\theta)\, \cos(\theta) + F}{M + m\, \sin^2(\theta)} \tag{2.7}$$

De plus, on peut récrire l'équation 2.4 sous la forme :

$$\ddot{\theta} = \frac{g\ \sin(\theta) - \ddot{x}\ \cos(\theta)}{l} \tag{2.8}$$

En remplaçant la relation 2.7 dans la relation 2.8, et en regroupant convenablement les termes similaires, on obtient :

$$\ddot{\theta} = \frac{(M+m)\, g\ \sin(\theta) - m\, l\, \dot{\theta}^2\ \sin(\theta)\ \cos(\theta) - F\ \cos(\theta)}{l\,(M + m\ \sin^2(\theta))} \tag{2.9}$$

Ainsi, les équations 2.7 et 2.9 forment le système d'équations différentielles ordinaires qui décrivent le mouvement du pendule inversé (dans les hypothèses sus-mentionnées). En résolvant ce système, on obtient les fonctions d'une seule variable $\theta(t)$ (qui nous donne la variation de l'angle θ formé par la tige du pendule avec la verticale en fonction du temps) et $x(t)$ (qui nous donne la variation de la position du chariot en fonction du temps), ainsi que leurs premières et deuxièmes dérivées.

2.2 Résolution numérique du système d'équations différentielles ordinaires du modèle

Vu que le système d'équations différentielles ordinaires mentionné ci-dessus est obtenu à partir d'un problème mécanique concret, l'existence de ses solutions (pour des conditions initiales "raisonnables") est normalement assurée par la nature physique du phénomène étudié. Quant à l'unicité de ses solutions, on suppose que l'étude reste déterministe (et, en l'absence de perturbations externes intentionnelles, sans aspects chaotiques).

Il convient de remarquer que le système d'équations 2.7 et 2.9 est un système de deux équations différentielles ordinaires d'ordre deux qui peut être transformé dans un système équivalent de quatre équations différentielles ordinaires d'ordre un. Plus précisément, on considère la vitesse du chariot $\dot{x}(t)$ et la vitesse angulaire du pendule $\dot{\theta}(t)$ comme deux nouvelles fonctions inconnues notées $x_1(t)$ et, respectivement, $\theta_1(t)$. Ainsi, le nouveau système équivalent devient :

$$\begin{cases} \dot{x} = x_1 \\ \dot{\theta} = \theta_1 \\ \dot{x_1} = \dfrac{m\, l\, \theta_1^2\ \sin(\theta) - m\, g\ \sin(\theta)\ \cos(\theta) + F}{M + m\ \sin^2(\theta)} \\ \dot{\theta_1} = \dfrac{(M+m)\, g\ \sin(\theta) - m\, l\, \theta_1^2\ \sin(\theta)\ \cos(\theta) - F\ \cos(\theta)}{l\,(M + m\ \sin^2(\theta))} \end{cases} \tag{2.10}$$

De cette manière (voir, par exemple, [6]), pour résoudre numériquement le système d'EDO ci-dessus, il faut résoudre numériquement des équations différentielles ordinaires d'ordre un de la forme générale suivante :

$$\begin{cases} \dot{y}(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases} \tag{2.11}$$

où t appartient à un intervalle réel borné $[t_0, T]$, avec t_0 et T le moment du début et, respectivement, de la fin de l'étude, et y_0 précise la valeur initiale de la fonction inconnue $y(t)$.

La résolution numérique du problème 2.11 (connu sous le nom du problème de Cauchy) commence par la discrétisation de l'intervalle $T-t_0$ d'étude à l'aide d'une succession strictement croissante de moments (nœuds) de calcul :

$$t_0 < t_1 < ... < t_n < t_{n+1} < ... < t_N = T \tag{2.12}$$

Ensuite, sur chaque sous-intervalle $[t_n, t_{n+1}]$, on intègre la première équation du problème 2.11, i.e. l'équation différentielle à résoudre, à l'aide du théorème fondamental du calcul intégral. Ainsi, vu que $\dot{y}(t) = \dfrac{dy}{dt}$, on obtient :

$$\int_{t_n}^{t_{n+1}} dy = \int_{t_n}^{t_{n+1}} f(t, y(t))dt \tag{2.13}$$

ou encore :

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t, y(t))dt \tag{2.14}$$

où $y_n = y(t_n)$ et $y_{n+1} = y(t_{n+1})$.

Il reste maintenant à calculer l'intégrale définie intervenant dans le membre de droite de la relation ci-dessus à l'aide d'une formule d'intégration (de quadrature) numérique adéquate, à savoir la formule dite du "point de gauche" ou du "rectangle à gauche". Ceci revient à approximer la valeur de l'intégrale $\int_{t_n}^{t_{n+1}} f(t, y(t))dt$ par l'aire (corrigée avec le "bon" signe) d'un rectangle de base égale à la longueur de l'intervalle d'intégration $t_{n+1} - t_n$ et la hauteur égale à la valeur de la fonction à intégrer au "point de gauche" $f(t_n, y(t_n))$. Par conséquent :

$$\int_{t_n}^{t_{n+1}} f(t, y(t))dt \approx (t_{n+1} - t_n) f(t_n, y(t_n)) \tag{2.15}$$

Finalement, si on note par $\widetilde{y}_n$ et $\widetilde{y}_{n+1}$ les valeurs approchées numériquement de $y(t_n)$ et $y(t_{n+1})$, on obtient le schéma numérique suivant :

$$\begin{cases} \widetilde{y}_{n+1} = \widetilde{y}_n + (t_{n+1} - t_n) f(t_n, \widetilde{y}_n) \\ \widetilde{y}_0 = y_0 \end{cases} \tag{2.16}$$

Il convient de mentionner que le schéma numérique 2.16 obtenu pour résoudre le problème 2.11 est connu sous le nom de schéma d'*Euler progressif* qui est un schéma explicite à un pas.

L'implémentation dans le code Java de l'approche numérique présentée ci-dessus sera détaillée dans le chapitre 3.3.

2.3 Considérations sur le système dynamique qui décrit le mouvement du pendule inversé

2.3.1 Modèle linéaire d'état (Linéarisation des équations)

On peut récrire le système d'EDO 2.10 sous la forme matricielle suivante :

$$\begin{pmatrix} \dot{x} \\ \dot{\theta} \\ \dot{x_1} \\ \dot{\theta_1} \end{pmatrix} = \begin{pmatrix} x_1 \\ \theta_1 \\ \dfrac{m\,l\,\theta_1^2\,\sin(\theta) - m\,g\,\sin(\theta)\,\cos(\theta)}{M + m\,\sin^2(\theta)} \\ \dfrac{(M+m)\,g\,\sin(\theta) - m\,l\,\theta_1^2\,\sin(\theta)}{l\,(M + m\,\sin^2(\theta))} \end{pmatrix} + \frac{1}{l(M + m\,\sin^2(\theta))} \cdot \begin{pmatrix} 0 \\ 0 \\ l \\ -\cos(\theta) \end{pmatrix} \cdot F \tag{2.17}$$

Le système ci-dessus formé de quatre équations différentielles ordinaires de premier ordre peut être vu comme un *système dynamique* avec n=4 *variables d'état* ou, tout simplement, *états*, à savoir $x(t)$, $\theta(t)$, $x_1(t)$ et $\theta_1(t)$, et q=1 i.e. une *commande* ou *entrée*, à savoir la force motrice $F(t)$. Un tel système d'EDO est couramment appelé *modèle d'état* ou *forme d'état* (voir par exemple [7] ou [8]).

Il convient de remarquer que le système d'EDO 2.17 est un système *stationnaire* car les membres de droite de ses équations ne dépendent pas explicitement du temps.

En général, le but de l'étude d'un système dynamique est de calculer l'évolution dans le temps de certaines grandeurs appelées *sorties* ou *mesures*. Dans notre cas, les sorties sont simplement l'angle de déviation du pendule $\theta(t)$ et la position du chariot $x(t)$.

Afin de résoudre le système d'EDO 2.17, il faut connaître non seulement les conditions initiales pour les variables d'état mais aussi la variation de la commande $F(t)$ à chaque moment t. En fait, notre but est de **contrôler** le

comportement du système dynamique 2.17 en agissant sur les variables d'état par l'intermédiaire de la commande $F(t)$.

Plus précisément, la stabilisation du mouvement du pendule inversé autour de sa position d'équilibre instable sera possible grâce à une force motrice appropriée agissant sur le chariot.

Si la variation de la commande $F(t)$ est déjà fixée au début de la simulation, il s'agit là d'une simulation en *boucle ouverte*.

Par contre, si la commande F est calculée durant la simulation, de manière dynamique, en fonction de l'évolution des variables d'état, il s'agit là d'une simulation en *boucle fermée*. Plus précisément, dans ce deuxième cas, afin de choisir convenablement l'entrée $F(t)$, on utilise une *rétroaction* ou *feedback* (ou encore *retour de sortie*) de la forme générale :

$$F(t) = \Phi(x(t), \theta(t), t) \tag{2.18}$$

D'ailleurs, la simulation numérique présentée dans le chapitre 3 donne à l'utilisateur la possibilité de choisir entre plusieurs variantes pour stabiliser le pendule. De plus, grâce à l'implémentation du design pattern *Strategy*, la solution informatique proposée permet aussi au programmeur de définir plusieurs stratégies pour le contrôle du mouvement du pendule inversé.

Le système d'EDO 2.17 est non linéaire et sa résolution analytique n'est pas possible (du moins pour le moment). Par conséquent, la dynamique du mouvement du pendule inversé sous le contrôle de la variable de commande F sera obtenue de manière numérique.

Cependant, il est intéressant de pouvoir aussi faire une *étude qualitative* du système d'EDO obtenu suite à la linéarisation du système non linéaire 2.17. Pour cela, on considère d'abord le *système libre* associé au système dynamique 2.17, c'est-à-dire le système d'EDO qui résulte si $F(t) = 0$ (i.e. si aucune commande ne contrôle le mouvement du pendule inversé) :

$$\begin{pmatrix} \dot{x} \\ \dot{\theta} \\ \dot{x_1} \\ \dot{\theta_1} \end{pmatrix} = \begin{pmatrix} x_1 \\ \theta_1 \\ \dfrac{m\,l\,\theta_1^2\,\sin(\theta) - m\,g\,\sin(\theta)\,\cos(\theta)}{M + m\,\sin^2(\theta)} \\ \dfrac{(M+m)\,g\,\sin(\theta) - m\,l\,\theta_1^2\,\sin(\theta)}{l\,(M + m\,\sin^2(\theta))} \end{pmatrix} \tag{2.19}$$

Supposons qu'au moment initial $t_{in} = 0$, le pendule inversé se trouve dans la position d'équilibre instable caractérisée par $\theta(0) = \theta_0 = 0$ et que sa vitesse angulaire initiale est nulle $\theta_1(0) = \dot{\theta}(0) = 0$.

Alors, grâce au système libre 2.19, on constate qu'à tout moment ultérieur $t > 0$, le pendule reste dans la position d'équilibre instable ($\theta(t) = 0$ et

$\theta_1(t) = \dot{\theta}(t) = 0$). De plus, l'accélération du chariot qui était initialement nulle $\dot{x}_1(0) = \ddot{x}(0) = 0$ reste aussi nulle par la suite $\dot{x}_1(t) = \ddot{x}(t) = 0$.

Par conséquent, si le chariot était au repos au moment initial $x_1(0) = \dot{x}(0) = 0$, il reste au repos aussi par la suite $x_1(t) = \dot{x}(t) = 0$, ou si le chariot avait une vitesse initiale non nulle $x_1(0) = \dot{x}(0) \neq 0$, il se déplace par la suite avec cette même vitesse constante $x_1(t) = \dot{x}(t) = \dot{x}(0)$, pour $\forall t > 0$.

On peut ainsi conclure que l'état correspondant à la position d'équilibre instable est un **état stationnaire** pour le système libre 2.19 (dans le sens où si le pendule se trouve initialement dans cet état, il y reste pour tout moment ultérieur).

Par la suite, on va essayer d'obtenir une forme plus simple pour le système dynamique 2.17 grâce à la linéarisation de ses équations autour de l'état stationnaire évoqué ci-dessus. L'expression approchée ainsi obtenue est appelée couramment **modèle linéaire d'état** (ou *système linéarisé tangent*).

Concrètement, on considère de petites variations $\varepsilon(t)$ de l'angle $\theta(t)$ autour de la position d'équilibre instable $\theta_0 = 0$. Autrement dit, on suppose qu'à tout moment t, on peut écrire :

$$\theta(t) = \theta_0 + \varepsilon(t) = \varepsilon(t) \tag{2.20}$$

où :

$$|\varepsilon(t)| << 1, \forall t. \tag{2.21}$$

En dérivant une fois la relation 2.20, on obtient :

$$\theta_1(t) = \dot{\theta}(t) = \dot{\varepsilon}(t) \tag{2.22}$$

De plus, on peut aussi supposer que :

$$\varepsilon^2(t) \approx 0 \tag{2.23}$$

et que :

$$\dot{\varepsilon}^2(t) \approx 0 \tag{2.24}$$

En outre, comme la fonction $\cos(\theta)$ est suffisamment régulière (car elle est en fait de classe C^∞), on peut approximer sa valeur pour de petits angles θ en écrivant la formule de Taylor (par rapport à θ) autour du point d'équilibre instable $\theta_0 = 0$ et en gardant seulement la partie linéaire de l'expression ainsi obtenue :

$$\begin{aligned} \cos(\theta) &\approx \cos(\theta_0) + \left.\frac{d(\cos\theta)}{d\theta}\right|_{\theta=\theta_0} \cdot (\theta - \theta_0) \\ &= cos(0) - sin(0) \cdot \varepsilon \\ &= 1 \end{aligned} \tag{2.25}$$

De même, pour la fonction $\sin(\theta)$:

$$\begin{aligned}\sin(\theta) &\approx \sin(\theta_0) + \left.\frac{d(\sin\theta)}{d\theta}\right|_{\theta=\theta_0} \cdot (\theta - \theta_0) \\ &= sin(0) + cos(0) \cdot \varepsilon \\ &= \varepsilon\end{aligned} \tag{2.26}$$

En remplaçant les relations de 2.22 à 2.26 dans le système 2.17, on obtient :

$$\begin{pmatrix}\dot{x}\\ \dot{\theta}\\ \dot{x_1}\\ \dot{\theta_1}\end{pmatrix} = \begin{pmatrix}x_1\\ \dot{\varepsilon}\\ \dfrac{-m\,g\,\varepsilon}{M}\\ \dfrac{(M+m)\,g\,\varepsilon}{l\,M}\end{pmatrix} + \frac{1}{lM}\cdot\begin{pmatrix}0\\0\\l\\-1\end{pmatrix}\cdot F \tag{2.27}$$

Mais, vu les relations 2.20 et 2.22, on peut récrire tout simplement :

$$\begin{pmatrix}\dot{x}\\ \dot{\theta}\\ \dot{x_1}\\ \dot{\theta_1}\end{pmatrix} = \begin{pmatrix}x_1\\ \theta_1\\ \dfrac{-m\,g\,\theta}{M}\\ \dfrac{(M+m)\,g\,\theta}{l\,M}\end{pmatrix} + \frac{1}{lM}\cdot\begin{pmatrix}0\\0\\l\\-1\end{pmatrix}\cdot F \tag{2.28}$$

Le système ci-dessus représente le système d'EDO *linéaire* obtenu suite à la linéarisation du système 2.17 autour de l'état d'équilibre instable (caractérisé par $\theta_0 = 0$).

2.3.2 Étude analytique : réglage en boucle fermée par commande proportionnelle

Avant de s'occuper d'une manière plus générale de la stabilité et de la commandabilité du système dynamique 2.28, on peut d'abord revenir aux variables x et θ et récrire seulement les deux dernières équations différentielles ordinaires sous la forme :

$$\ddot{x} = -\frac{mg}{M}\cdot\theta + \frac{1}{M}\cdot F \tag{2.29}$$

$$\ddot{\theta} = \frac{(M+m)g}{lM}\cdot\theta - \frac{1}{lM}\cdot F \tag{2.30}$$

Afin d'essayer de stabiliser le pendule inversé, il suffit maintenant d'utiliser un réglage en boucle fermée en choisissant comme commande une force

proportionnelle à la "déviation" du pendule inversé par rapport à la position d'équilibre instable (qui correspond à la verticale), c'est-à-dire :

$$F(t) = k \cdot \theta(t) \tag{2.31}$$

où k est une *constante de proportionnalité* (ou de rappel) convenablement choisie (et dont la dimension est $\frac{N}{rad}$ ou, tout simplement N).

Étant donné les sens positifs choisis pour l'angle θ et pour l'abscisse x, la constante de proportionnalité k doit être positive $k > 0$.

En remplaçant 2.31 en 2.30, on obtient l'équation qui gouverne le mouvement du pendule inversé :

$$\ddot{\theta} = \frac{(M+m)g - k}{lM} \cdot \theta \tag{2.32}$$

Or, l'équation différentielle d'ordre deux 2.32 accepte des solutions périodiques de la forme :

$$\theta(t) = \theta_{max} \cdot \cos(\omega t) \tag{2.33}$$

où $\theta_{max} > 0$ est l'amplitude et ω est la pulsation de ces oscillations.

Afin de déterminer l'expression de ω, on calcule successivement :

$$\dot{\theta} = -\omega \cdot \theta_{max} \cdot \sin(\omega t) \tag{2.34}$$

$$\ddot{\theta} = -\omega^2 \cdot \theta_{max} \cdot \cos(\omega t) \tag{2.35}$$

et on remplace les relations 2.33 et 2.35 dans l'équation différentielle 2.32 :

$$-\omega^2 \cdot \theta_{max} \cdot \cos(\omega t) = \frac{(M+m)g - k}{lM} \cdot \theta_{max} \cdot \cos(\omega t) \tag{2.36}$$

Finalement, on obtient :

$$\omega = \sqrt{\frac{k - (M+m) \cdot g}{lM}} \tag{2.37}$$

Il convient de remarquer que cette dernière relation met en évidence une contrainte à remplir par la commande F. Plus précisément, afin d'assurer une pulsation réelle et strictement positive $\omega > 0$, la constante de rappel k doit satisfaire la condition :

$$k > (M+m) \cdot g \tag{2.38}$$

et cette information pourra être intégrée dans la stratégie de stabilisation du pendule inversé.

Quant à l'amplitude θ_{max}, elle peut être calculée en fonction de la façon dont le mouvement du pendule démarre au voisinage de la position d'équilibre instable à $t_{in} = 0$.

Il convient de remarquer que le choix de la fonction périodique cos (au détriment de la fonction sin) dans la relation 2.33 permet (voire impose) les conditions initiales suivantes :

- pour l'angle θ :
$$\theta(t_{in}) = \theta(0) = \theta_{in} \tag{2.39}$$
où θ_{in} est l'angle correspondant à la déviation initiale du pendule par rapport à la verticale ;
- pour la vitesse angulaire $\dot{\theta}$:
$$\dot{\theta}(t_{in}) = \dot{\theta}(0) \overset{2.34}{=} -\omega \cdot \theta_{max} \cdot \sin(0) = 0 \tag{2.40}$$

Le fait qu'il n'y a pas de déphasage initial dans la relation 2.33 implique que l'angle initial doit être positif $\theta_{in} > 0$, mais ceci ne restreint pas la généralité de l'approche.

En fait, un démarrage du mouvement avec $\theta_{in} < 0$ reviendrait à chercher des solutions de la forme :

$$\theta(t) = \theta_{max} \cdot \cos(\omega t + \pi) \tag{2.41}$$

ou encore :

$$\theta(t) = -\theta_{max} \cdot \cos(\omega t) \tag{2.42}$$

Par la suite, on garde la relation 2.33 qui nous permet de calculer :

$$\theta(0) = \theta_{max} \cdot \cos(0) = \theta_{max} \tag{2.43}$$

Les relations 2.39 et 2.43 donnent finalement l'amplitude recherchée :

$$\theta_{max} = \theta_{in} \tag{2.44}$$

De plus, vu que $\omega = \dfrac{2\pi}{T}$, la période correspondante T a l'expression :

$$T = 2\pi \cdot \sqrt{\frac{lM}{k - (M+m)g}} \tag{2.45}$$

Finalement, le mouvement oscillatoire du pendule inversé peut être décrit par la relation suivante :

$$\theta(t) = \theta_{in} \cdot \cos\left(\sqrt{\frac{k - (M+m)g}{lM}} \cdot t\right) \tag{2.46}$$

Quant au chariot, on remplace les relations 2.31 et 2.33 dans 2.29 et on obtient l'équation qui gouverne son mouvement :

$$\ddot{x} = \frac{k - mg}{M} \cdot \theta_{max} \cdot \cos(\omega t) \tag{2.47}$$

À son tour, l'équation différentielle ordinaire d'ordre deux ci-dessus accepte des solutions périodiques de même pulsation que celle du pendule et de la forme suivante :

$$x(t) = x_{max} \cdot \cos(\omega t + \varphi_0) \tag{2.48}$$

où $x_{max} > 0$ est l'amplitude et φ_0 est le déphasage initial de ces oscillations.

Vu les sens positifs choisis pour l'angle θ, pour l'abscisse x ainsi que les relations 2.39 et 2.40, la condition initiale pour la position du chariot doit être :

$$x(0) = -x_{max} \tag{2.49}$$

En remplaçant la relation 2.48 en 2.49, on obtient :

$$x_{max} \cdot \cos(\varphi_0) = -x_{max} \tag{2.50}$$

ou encore :

$$\varphi_0 = \pi \tag{2.51}$$

Vu que :

$$\cos(\omega t + \varphi_0) \overset{2.51}{=} \cos(\omega t + \pi) = -\cos(\omega t) \tag{2.52}$$

la relation 2.48 devient :

$$x(t) = -x_{max} \cdot \cos(\omega t) \tag{2.53}$$

En procédant comme pour l'angle θ, on calcule :

$$\dot{x} = \omega \cdot x_{max} \cdot \sin(\omega t) \tag{2.54}$$

$$\ddot{x} = \omega^2 \cdot x_{max} \cdot \cos(\omega t) \tag{2.55}$$

En remplaçant les relations 2.53 et 2.55 dans l'équation différentielle 2.47, on obtient :

$$\omega^2 \cdot x_{max} \cdot \cos(\omega t) = \frac{k - mg}{M} \cdot \theta_{max} \cdot \cos(\omega t) \tag{2.56}$$

ou encore :

$$\begin{aligned} x_{max} &= \frac{k - mg}{M\omega^2} \cdot \theta_{max} \\ &\overset{2.37}{=} \frac{k - mg}{M} \cdot \frac{M}{k - (M + m)g} \cdot l \cdot \theta_{max} \\ &= \frac{(k - mg)}{k - (M + m)g} \cdot l \cdot \theta_{in} \end{aligned} \tag{2.57}$$

Cette dernière relation nous donne l'amplitude du mouvement oscillatoire du chariot et il convient de remarquer que, vu la condition 2.38, la valeur obtenue est bien strictement positive $x_{max} > 0$.

Finalement, les oscillations périodiques effectuées par le chariot afin de stabiliser le pendule inversé correspondent à la relation suivante :

$$x(t) = -\frac{(k - mg)}{k - (M + m)g} \cdot l \cdot \theta_{in} \cdot \cos\left(\sqrt{\frac{k - (M + m)g}{lM}} \cdot t\right) \tag{2.58}$$

Les résultats obtenus suite à l'étude analytique que l'on vient de finir peuvent être regroupés sur le schéma ci-dessous (Figure 2.5) où x_{eq} indique la position du chariot correspondant à l'équilibre instable, $\dot{x}_{max}$ est la vitesse maximale (positive) du chariot, $\dot{\theta}_{max}$ est la vitesse angulaire maximale (positive) du pendule et p est un entier positif quelconque ($p \in \mathbb{N}$).

Durant une période T, l'ensemble formé par le pendule et le chariot passe successivement par les états $a) \rightarrow b) \rightarrow c) \rightarrow d) \rightarrow e)$.

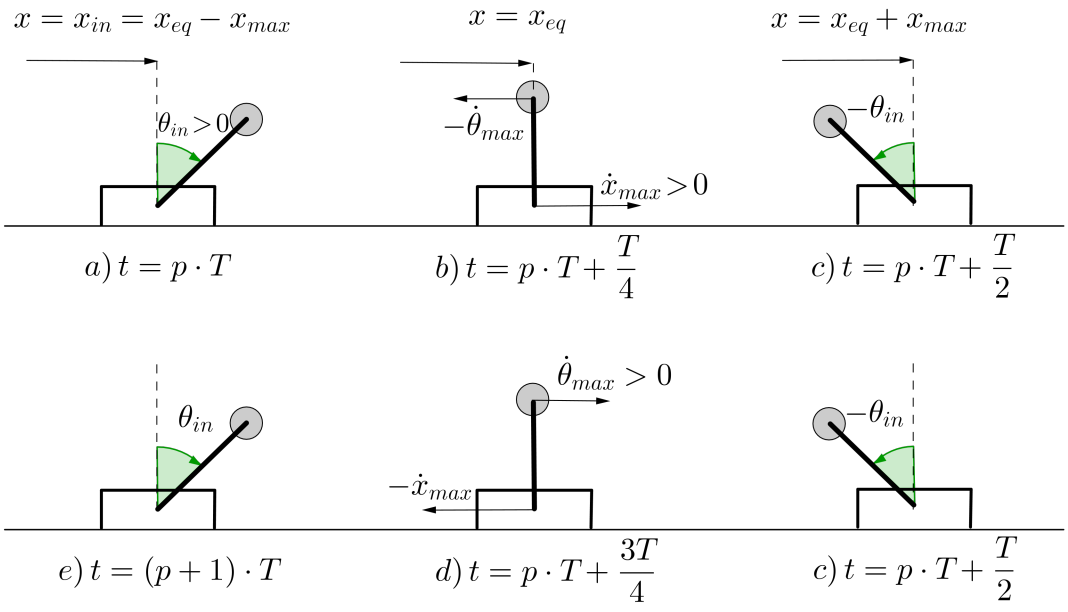


FIGURE 2.5 – Étude analytique : le mouvement durant une période

2.3.3 Considérations qualitatives sur la stabilisation du pendule inversé

En reprenant le système linéaire 2.28, on introduit le vecteur colonne $X(t)$ dont les composantes sont les variables d'état :

$$X(t) = (x(t), \theta(t), x_1(t), \theta_1(t))^T \tag{2.59}$$

Ainsi, le système 2.28 devient :

$$\dot{X}(t) = A \cdot X(t) \; + \; B \cdot F(t) \tag{2.60}$$

où :

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -\dfrac{m}{M}g & 0 & 0 \\ 0 & \dfrac{(M+m)g}{lM}g & 0 & 0 \end{pmatrix} \tag{2.61}$$

$$B = \begin{pmatrix} 0 & 0 & \dfrac{1}{M} & -\dfrac{1}{lM} \end{pmatrix}^T \tag{2.62}$$

Si on tient compte du fait que les grandeurs qui nous intéressent durant la simulation du mouvement du pendule inversé sont l'angle $\theta(t)$ et la position du chariot $x(t)$, on peut introduire le vecteur colonne des sorties (ou des mesures) noté $Y(t)$:

$$Y(t) = (x(t), \theta(t))^T \tag{2.63}$$

Ainsi, il convient d'ajouter à l'équation 2.60 une nouvelle équation matricielle qui relie les sorties avec les variables d'état :

$$Y(t) = D \cdot X(t) \tag{2.64}$$

où

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{2.65}$$

Par la suite, on s'intéresse à la *stabilité locale* de l'état d'équilibre du *système libre* autour duquel le modèle linéaire a été obtenu pour le pendule inversé.

D'une manière informelle mais assez intuitive, on peut dire qu'un état d'équilibre est **stable** si de petites variations (ou perturbations) initiales des variables d'état ne produisent que de petits écarts pour tout moment ultérieur.

Autrement dit, pour $\forall t > 0$, les valeurs des variables d'état restent proches des valeurs initiales correspondantes à $t_{in} = 0$ si ces valeurs initiales sont proches, à leur tour, de l'état d'équilibre.

Si, *de plus*, à partir des valeurs initiales proches de l'état d'équilibre le système évolue (plus ou moins vite) vers l'état d'équilibre, cet état d'équilibre est nommé **asymptotiquement stable**.

Bien évidemment, un état d'équilibre qui n'est pas stable est dit **instable**.

Afin de connaître la stabilité du modèle linéaire obtenu pour le pendule inversé, on utilise la propriété suivante, valable pour un système linéaire libre de la forme $\dot{X} = AX$ (selon [9], page 22) :

" Le système linéaire est :

- ***instable*** *si au moins une valeur propre de A est à partie réelle strictement positive ;*
- ***stable*** *si toutes les valeurs propres de A sont à partie réelle négative ;*
- ***asymptotiquement stable*** *si toutes les valeurs propres de A sont à partie réelle strictement négative. "*

Par conséquent, on calcule les valeurs propres λ_i, $i \in \{1, 2, 3, 4\}$, de la matrice A, c'est-à-dire les racines du polynôme caractéristique :

$$det(A - \lambda I) = 0 \tag{2.66}$$

où I est la matrice unité (ou identité) d'ordre 4.

Ainsi :

$$\begin{vmatrix} -\lambda & 0 & 1 & 0 \\ 0 & -\lambda & 0 & 1 \\ 0 & -\frac{m}{M}g & -\lambda & 0 \\ 0 & \frac{(M+m)g}{lM}g & 0 & -\lambda \end{vmatrix} = 0 \tag{2.67}$$

ce qui donne :

$$\lambda^4 - \lambda^2 \cdot \frac{(M+m)g}{lM} = 0$$

ou encore :

$$\lambda^2 \cdot \left(\lambda^2 - \frac{(M+m)g}{lM} \right) = 0 \tag{2.68}$$

L'équation algébrique ci-dessus accepte quatre solutions *réelles*, à savoir :

$$\begin{cases} \lambda_1 = \lambda_2 = 0 \\ \lambda_{3,4} = \pm\sqrt{\frac{(M+m)g}{lM}} \end{cases} \tag{2.69}$$

Vu qu'une des valeurs propres de la matrice A est bien *réelle* et *positive*, on peut conclure que le modèle linéaire obtenu pour le pendule inversé est bien **instable**.

Après avoir étudié sa stabilité, il convient maintenant d'étudier la *commandabilité* de ce modèle linéaire d'état.

D'une manière générale informelle mais assez intuitive, un système dynamique (non linéaire ou linéaire) est *commandable* (en un temps $\tau > 0$) si pour tout couple d'états on peut trouver une commande en boucle ouverte qui amène le système du premier état au deuxième état (en temps τ).

Pour le cas des systèmes linéaires, la commandabilité peut être déterminée grâce au critère de Kalman (voir, par exemple, [10], Théorème 10) qui, avec nos notations, précise que : *" Le système $\dot{X} = AX + BF$ est commandable* si et seulement si *la matrice de commandabilité $C = (B|AB|...|A^{n-1}B)$ est de rang $n = dim(X)$. "*

Pour le cas du pendule inversé, la matrice A est de taille $n \times n = 4 \times 4$, la matrice B est de taille $n \times q = 4 \times 1$ et la matrice C est de taille $n \times (n \cdot q) = 4 \times 4$ (car n est le nombre de variables d'état et q est le nombre de commandes).

Donc, on calcule successivement les matrices A^2, A^3, AB, A^2B et A^3B et on obtient :

$$A^2 = \begin{pmatrix} 0 & -\frac{m}{M}g & 0 & 0 \\ 0 & \frac{(M+m)g}{lM} & 0 & 0 \\ 0 & 0 & -\frac{m}{M}g & 0 \\ 0 & 0 & 0 & \frac{(M+m)g}{lM} \end{pmatrix} \tag{2.70}$$

$$A^3 = \begin{pmatrix} 0 & 0 & 0 & -\frac{m}{M}g \\ 0 & 0 & 0 & \frac{(M+m)g}{lM} \\ 0 & -\frac{m(M+m)g^2}{lM^2} & 0 & 0 \\ 0 & 0 & 0 & \frac{(M+m)^2g^2}{l^2M^2} \end{pmatrix} \tag{2.71}$$

$$A \cdot B = \begin{pmatrix} \frac{1}{M} & -\frac{1}{lM} & 0 & 0 \end{pmatrix}^T \tag{2.72}$$

$$A^2 \cdot B = \begin{pmatrix} 0 & 0 & \frac{mg}{lM^2} & -\frac{(M+m)g}{l^2M^2} \end{pmatrix}^T \tag{2.73}$$

$$A^3 \cdot B = \begin{pmatrix} \frac{mg}{lM^2} & -\frac{(M+m)g}{l^2M^2} & 0 & 0 \end{pmatrix}^T \tag{2.74}$$

Finalement, la matrice C devient :

$$\begin{aligned} C &= (B|AB|A^2B|A^3B) \\ &= \begin{pmatrix} 0 & \frac{1}{M} & 0 & \frac{mg}{lM^2} \\ 0 & -\frac{1}{lM} & 0 & -\frac{(M+m)g}{l^2M^2} \\ \frac{1}{M} & 0 & \frac{mg}{lM^2} & 0 \\ -\frac{1}{lM} & 0 & -\frac{(M+m)g}{l^2M^2} & 0 \end{pmatrix} \end{aligned} \tag{2.75}$$

Après des calculs standards, le déterminant de la matrice C vaut :

$$det(C) = -\frac{g^2}{l^4M^4} \tag{2.76}$$

Vu que $det(C) \neq 0$, il s'en suit que :

$$rang(C) = 4 = dim(X) \tag{2.77}$$

et, grâce au critère de Kalman mentionné auparavant, on peut conclure que le système linéaire d'EDO associé à l'étude du pendule inversé est **commandable**.

Autrement dit, pour deux états quelconques du modèle linéaire, on peut trouver et fixer à l'avance une loi de variation de la commande $F(t)$ (donc de la force qui entraîne le chariot) de sorte que si la simulation du pendule inversé démarre dans un des deux états, alors elle arrive (grâce à cette commande en boucle ouverte) dans l'autre état.

Par la suite, la commandabilité pourra être résolue à l'aide d'une méthode appropriée (par exemple, en utilisant la forme normale dite de Brunowski ou par un autre contrôle en boucle ouverte) mais cet aspect ne fait pas partie des objectifs de ce travail (ni de mon domaine de compétences).

Chapitre 3

Simulateur du pendule inversé

Grâce aux résultats obtenus aux chapitres précédents, on dispose maintenant de tous les éléments physiques et mathématiques nécessaires pour concevoir un simulateur du mouvement d'un pendule inversé. Dans ce chapitre, on présente l'implémentation d'un tel simulateur sous la forme d'un projet Java réalisé en utilisant l'environnement de développement intégré (IDE) Eclipse.

Le choix du langage Java n'est pas anodin. Selon le site de référence Tiobe [11], au mois de janvier 2017, le langage de programmation le plus utilisé dans le monde est bien le langage Java (et, depuis de longues années, Java occupe une position privilégiée auprès des programmeurs). Cependant, ce n'est pas la renommée de Java qui a été déterminante dans notre choix, mais surtout les qualités de ce langage qui est orienté objets et qui permet une programmation graphique et événementielle performante [12].

Vu que le code source est abondamment commenté et que la documentation (API) associée (obtenue à l'aide de [5]) est présentée dans l'annexe A, on se contente par la suite de passer en revue seulement les aspects essentiels de la partie informatique du travail.

3.1 Structure du code

On a accordé une grande importance à la modularisation du code qui est une condition essentielle non seulement pour la conception initiale mais aussi pour la maintenance et les développements ultérieurs d'un logiciel. Dans la FIGURE 3.1, on présente le diagramme de classes UML de l'ensemble du projet.

Le projet est structuré en quatre packages :

- le package *alina.sim* qui contient :

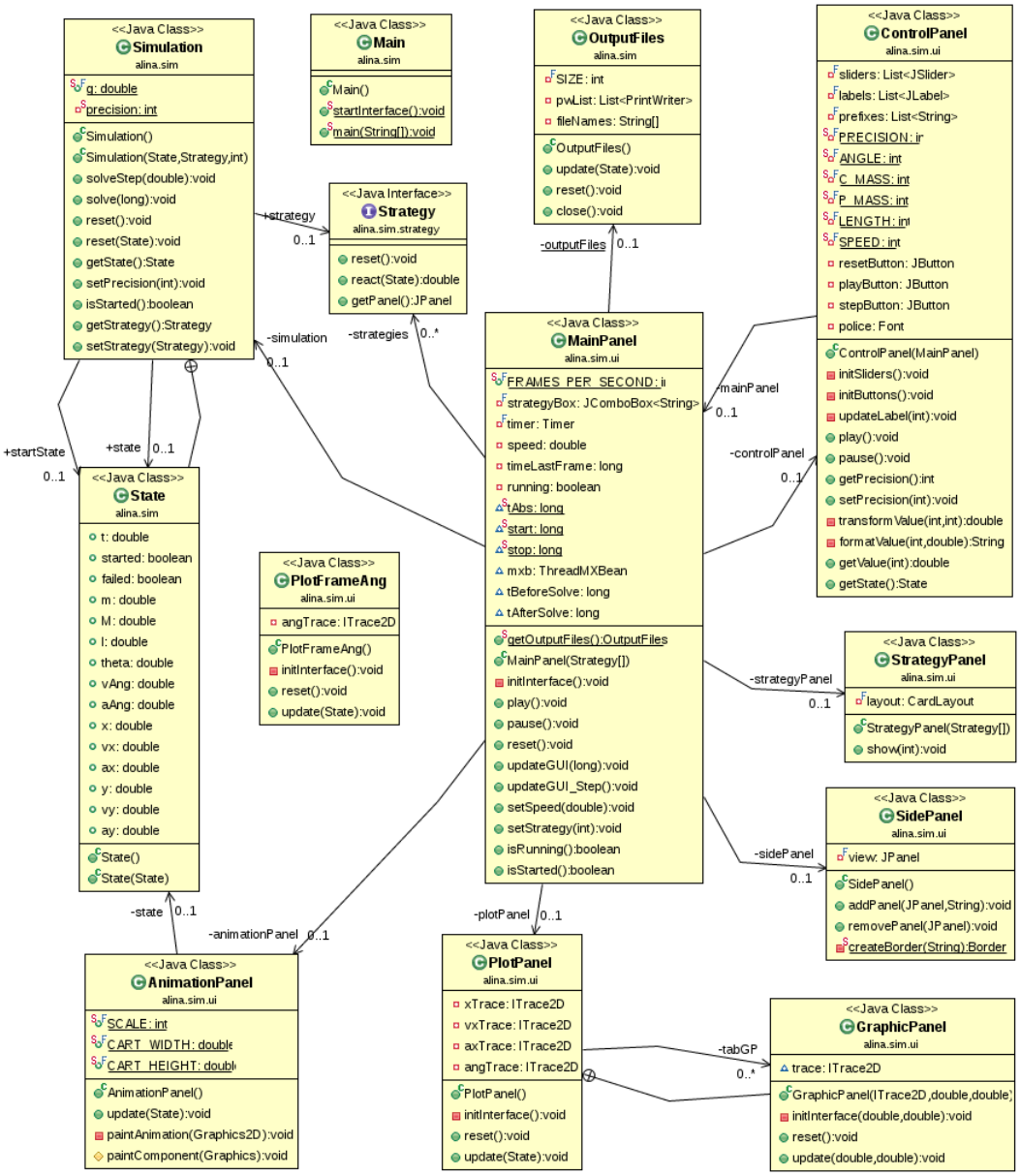


FIGURE 3.1 – Schéma UML du projet

- la classe principale *Main* où on trouve la méthode *main* qui est le point d'entrée de toute application Java autonome ; c'est dans cette méthode qu'on crée le container de premier niveau (top-level container) de type *JFrame* qui représente la fenêtre principale de notre interface graphique (GUI) ;
- la classe *Simulation* qui :

        - contient la classe interne *Simulation.State* dont les champs permettent de stocker toutes les informations pertinentes concernant l'état du système physique à un moment donné (par exemple : la position, la vitesse et l'accélération du chariot ou l'angle θ, la vitesse angulaire et l'accélération angulaire du pendule) ;
        - définit notamment la méthode *solveStep* qui utilise les informations sus-mentionnées afin de calculer et stocker le nouvel état du système après un laps (ou un pas) de temps donné et en fonction d'une certaine stratégie de stabilisation du pendule ; en fait, c'est dans cette méthode qu'on résout numériquement le système d'équations différentielles ordinaires qui gouvernent le mouvement du pendule inversé ;
    - la classe *OutputFiles* qui permet de stocker de manière "permanente" les résultats du calcul numérique ; plus précisément, les valeurs de la position, de la vitesse et de l'accélération du chariot, ainsi que de l'angle θ sont enregistrées, à chaque moment courant t, dans des fichiers texte (*.txt*) appropriés ; ainsi, après chaque simulation en temps réel, des données pertinentes restent disponibles et rendent possibles des post-traitements adéquats des cas étudiés ;
- le package *alina.sim.strategy* qui contient :
    - l'interface *Strategy* où on trouve surtout la méthode publique et abstraite *react* qui sera (re)définie dans les classes qui implémentent cette interface ; cette méthode reçoit en argument l'état d'une simulation (de type *Simulation.State*) et calcule et retourne la valeur de la force à appliquer sur le chariot afin de tenter de garder le pendule en équilibre ;
    - plusieurs classes (par exemple *SimpleStrategy* ou *ManualStrategy*) qui implémentent l'interface *Strategy* et qui précisent principalement la méthode redéfinie *react* ; cette architecture (qui correspond au design pattern *Strategy*) assure une approche versatile du problème et permet d'ajouter ou de changer facilement la façon de stabiliser le pendule inversé ;
- le package *alina.sim.ui* qui contient toutes les classes graphiques qui réalisent le design de l'interface GUI, interface qui est évidemment réactive aux actions de l'utilisateur ; il s'agit surtout du container intermédiaire *MainPanel* de type JPanel qui est placé dans le "content pane" du container de premier niveau *JFrame* ; le *MainPanel* contient à son tour d'autres containers intermédiaires (par exemple *AnimationPanel*, *PlotPanel*, *SidePanel*, etc.) qui contiennent à nouveau d'autres containers intermédiaires et/ou des éléments graphiques atomiques (par exemple des contrôles ou widgets de type *JButton*, *JSlider*, *JLa-*

bel, etc.) ; l'emboîtement de ces containers est présenté dans la FIGURE 3.2 ; plus précisément, le panneau *MainPanel* contient :

- un objet de type *AnimationPanel* qui sert à représenter le mouvement du chariot et du pendule inversé "en temps réel" (et qui utilise des classes standards des packages *java.awt* et *javax.swing*) ; à part l'ensemble chariot-tige-pendule, le panneau *AnimationPanel* affiche aussi les normes de la vitesse et de l'accélération du chariot sous forme de lignes horizontales dont les longueurs sont proportionnelles aux normes, ainsi que la norme de la vitesse angulaire du pendule sous la forme d'un arc de cercle de longueur proportionnelle à cette norme (avec, respectivement, les couleurs *cyan*, *pink* et *yellow*) ;
- un objet de type *PlotPanel* qui permet d'afficher graphiquement les variations en fonction du temps de certaines grandeurs physiques qui caractérisent le mouvement du pendule inversé (par exemple l'évolution dans le temps de la position, de la vitesse et de l'accélération du chariot ainsi que de l'angle θ du pendule) ; la classe *PlotPanel* utilise des bibliothèques de la famille *info.monitorenter* (par exemple la classe *Trace2DLtd* qui implémente l'interface *ITrace2D* et dont l'instanciation correspond à une courbe graphique ou la classe *Chart2D* dont l'instanciation correspond à une représentation graphique pouvant regrouper éventuellement plusieurs courbes) ;
- un objet de type *SidePanel* qui est un panneau abritant plusieurs autres containers intermédiaires, à savoir :
  - une instance de la classe *ControlPanel* qui regroupe, d'une part, plusieurs curseurs de type *JSlider* qui permettent à l'utilisateur final de choisir les principaux paramètres de la simulation (par exemple la masse du pendule, l'angle initial, la longueur de la tige, etc.) et, d'autre part, des boutons de type *JButton* qui permettent le contrôle du déroulement de la simulation (démarrage, exécution pas à pas, réinitialisation) ;
  - une instance de la classe standard *JPanel* qui contient une liste à choix de type *JComboBox* qui donne la possibilité à l'utilisateur de choisir la stratégie voulue pour la stabilisation du pendule ;
  - une instance de la classe *StrategyPanel* qui assure le réglage des paramètres pour une stratégie choisie ;

- le package *alina.uml* qui contient les éléments utiles à la réalisation du diagramme de classes du projet.

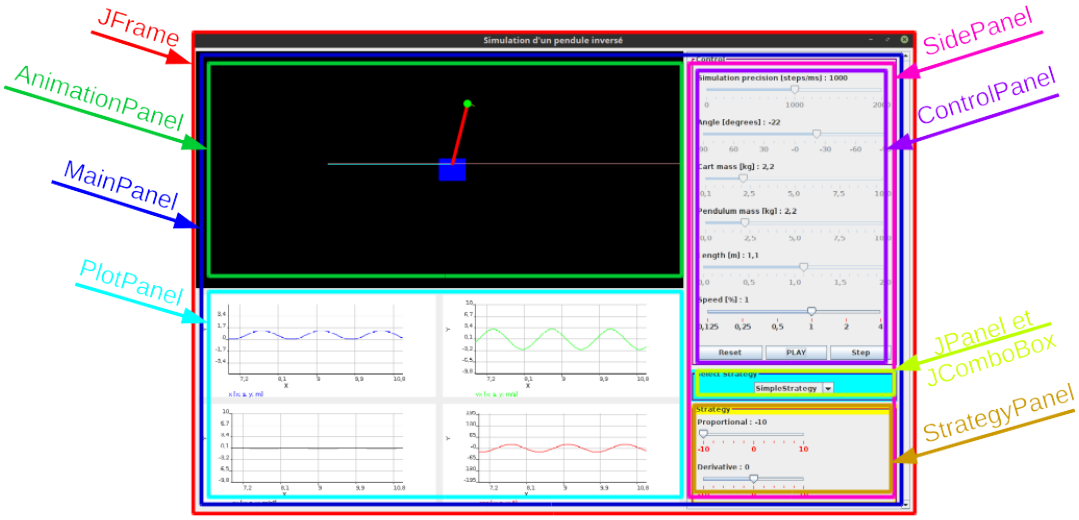


FIGURE 3.2 – Emboîtement des containers

3.2 Interface graphique utilisateur

Il convient de donner quelques détails concernant l'implémentation effective d'une simulation. En appuyant sur le bouton qui affiche initialement le texte `PLAY`, l'utilisateur peut démarrer une nouvelle simulation ou continuer une simulation arrêtée. En fait, une première fois appuyé, ce bouton change le texte affiché en `PAUSE` et permet ensuite d'interrompre la simulation en cours. Appuyé une deuxième fois, le bouton revient au texte `PLAY`, la simulation est reprise et ainsi de suite.

Le moteur de la simulation est le champ *timer* de la classe *MainPanel* qui est un objet (non graphique) de type *javax.swing.Timer* qui correspond à une sorte de "balise" qui émet des signaux (ou des événements) de type *ActionEvent*. À la création du *timer*, on indique :

- la période avec laquelle les signaux sont émis (et cette période correspond au champ static et final `FRAMES_PER_SECOND` - de la classe *MainPanel* - qui a la valeur par défaut de 50 cadres par seconde) ;
- l'objet écouteur de type *ActionListener* qui reçoit ces signaux.

C'est dans la méthode gestionnaire d'événements *actionPerformed* de cet écouteur que l'on indique ce qui doit se passer à l'envoi de chaque événement *ActionEvent*. Plus précisément, si la simulation est arrêtée (i.e. le champ booléen *running* de la classe *MainPanel* a la valeur *false*), les événements émis n'ont pas d'effet. Par contre, si la simulation est en cours (i.e. le champ booléen *running* de la classe *MainPanel* a la valeur *true*), chaque événement provoque un nouveau calcul des grandeurs physiques qui décrivent le mouve-

ment du pendule inversé (grâce à un appel de la méthode *solve* de la classe *Simulation*) ainsi que la mise à jour de l'animation (dans le panneau *AnimationPanel*) et des représentations graphiques (dans le panneau *PlotPanel*) qui s'en suivent.

3.3 Implémentation de la résolution numérique du système d'équations différentielles ordinaires

L'implémentation du calcul numérique présenté dans le chapitre 2.2 est réalisé par l'intermédiaire des étapes principales suivantes :

- chaque objet de type *Simulation* a un champ *state* de type classe interne *Simulation.State* ; ce champ est un objet qui contient, pour l'état courant :
  - les données "figées" du système mécanique (par exemple la masse du pendule m, la masse du chariot M, la longueur de la tige l, etc.) ;
  - le temps courant t ;
  - les valeurs courantes des données variables dans le temps (notamment la position horizontale x, la vitesse horizontale vx et l'accélération horizontale ax du chariot, ainsi que l'angle $theta$, la vitesse angulaire $vAng$ et l'accélération angulaire $aAng$ du pendule) ;
- afin de faire avancer la simulation (avec un intervalle de temps très petit dt que l'on accepte comme infinitésimal), c'est-à-dire afin d'obtenir le nouvel état du système mécanique (après l'intervalle dt), on appelle la méthode *solveStep*, en lui passant dt comme argument ; dans cette méthode, on procède ainsi :
  - on appelle la méthode *react* pour l'objet *strategy* (qui correspond à la stratégie choisie par l'utilisateur), en lui passant comme argument le champ *state* ; cette méthode retourne la nouvelle "bonne" valeur de la force de stabilisation du pendule fx en fonction des champs de l'objet *state* ; cette force peut être vue comme une force motrice produite par un moteur associé au chariot et qui lui permet d'accélérer ou de décélérer (voire de changer le sens du déplacement) afin de stabiliser le pendule ;
  - avec cette nouvelle valeur de la force, on calcule la nouvelle valeur de l'accélération ax et la nouvelle valeur de l'accélération $aAng$ (selon les formules obtenues grâce au principe de D'Alembert dans le chapitre 2 dédié à l'étude mécanique du pendule inversé) ;
  - à l'aide de la nouvelle valeur ax, on calcule la nouvelle vitesse hori-

zontale (par une instruction Java de la forme `vx = vx + ax * dt;`) et ensuite la nouvelle position du chariot (par une instruction Java de la forme `x = x + vx * dt;`) ;
    - à l'aide de la nouvelle valeur *aAng*, on calcule la nouvelle vitesse angulaire (par une instruction Java de la forme `vAng = vAng + aAng * dt;`) et ensuite le nouvel angle (par une instruction Java de la forme `theta = theta + vAng * dt;`) ;
- toutes ces nouvelles valeurs sont stockées dans les champs correspondants du champ *state* de l'objet *simulation* ; par conséquent, après l'appel de la méthode *solveStep*, le champ *state* correspond au nouvel état de la simulation ;
- en outre, chaque objet *simulation* a aussi un champ *precision* qui est initialisé par défaut à 10^3 et qui peut être modifié par l'utilisateur (par l'intermédiaire de l'interface graphique) ; en fait, la valeur du champ *precision* indique le nombre de sous-intervalles égaux utilisés dans le calcul numérique pour avancer d'une milliseconde ; ainsi, la valeur d'un laps *dt* exprimée en secondes et utilisée dans la méthode *solve* afin d'appeler la méthode *solveStep* sera `dt = 0.001/precision` et ceci revient à dire que, sans modification de la part de l'utilisateur, l'intervalle "infinitésimal" *dt* utilisé par défaut est de 10^{-6} secondes ;
- par contre, pour faire avancer la simulation à partir d'un moment courant t et durant un intervalle de temps plus conséquent Δt, on appelle la méthode *solve* en lui passant comme argument cet intervalle de temps exprimé en millisecondes ; c'est la méthode *solve* qui partage (discrétise) l'intervalle de temps à parcourir (à balayer) Δt en un nombre adéquat de petits sous-intervalles de valeurs égales *dt* et appelle ensuite la méthode *solveStep* pour chacun de ces sous-intervalles ; après l'appel de la méthode *solve*, le champ *state* de l'objet *simulation* correspond au nouvel état de la simulation pour le moment $t + \Delta t$.

De plus, par l'intermédiaire du champ *speed* de la classe *MainPanel* (qui peut être modifiée par l'utilisateur à l'aide de la GUI), la simulation peut être présentée :

- en temps réel (pour $speed = 1$) ;
- avec un déroulement au ralenti (pour $speed < 1$) ;
- avec un déroulement accéléré (pour $speed > 1$).

Il convient de donner quelques explications concernant l'échelle du temps utilisée, afin de préciser le rapport entre le temps "réel" intervenant dans les équations physiques déduites au chapitre 2 et le temps utilisé pour le déroulement de la simulation numérique. D'une manière générale, la valeur

du temps courant est stockée dans le champ *time* de l'objet *state* (de type *State*) qui est lui-même un champ de l'objet *simulation* (de type *Simulation*) qui définit chaque simulation effective.
Normalement, le moteur de l'animation (le champ *timer* de la classe *MainPanel*) émet des événements *ActionEvent* avec une périodicité constante (de 20 millisecondes car le champ static et final *FRAMES_PER_SECOND* vaut par défaut 50). Pour chaque tel signal, si la simulation est en cours, la méthode *actionPerformed* du gestionnaire d'événements associé au *timer* :

- demande, à l'événement reçu, le moment précis de son émission (son "timestamp" effectif) qui devient le temps courant et le stocke dans une variable locale *time* ;
- calcule l'intervalle de temps écoulé entre l'émission du dernier événement (disponible dans le champ *timeLastFrame*) et de l'événement courant (et sa valeur est normalement proche de la période du *timer*) ;
- appelle la méthode *updateGUI* en lui passant comme argument l'intervalle de temps ci-dessus ;
- copie la valeur du temps courant dans le champ *timeLastFrame*.

La méthode *updateGUI* de la classe *MainPanel* :

- appelle la méthode *solve* de la classe *Simulation* en lui passant comme paramètre son propre argument (qui est un intervalle de temps "réel") multiplié par la valeur du champ *speed* de la classe *MainPanel* ; ceci revient à une mise à l'échelle (rescaling) du temps car :
  - si $speed = 1$, la simulation se déroule en "temps réel" ;
  - si $speed > 1$ (par exemple 2), la simulation sera accélérée (dans cet exemple de 2 fois) ;
  - si $speed < 1$ (par exemple 0.25), la simulation sera ralentie (dans cet exemple de 4 fois) ;

  suite à l'appel de la méthode *solve*, des nouvelles valeurs pour les paramètres du système (y compris pour le temps mis éventuellement à l'échelle) sont calculées et stockées dans le champ *state* ;
- appelle la méthode *update* de la classe *PlotPanel* (en lui passant en argument le champ *state*), afin de mettre à jour les représentations graphiques associées à la simulation ;
- appelle la méthode *update* de la classe *AnimationPanel* (en lui passant en argument le champ *state*), afin de mettre à jour la représentation animée du chariot et du pendule inversé ;
- appelle la méthode *update* de la classe *OutputFiles* (en lui passant en argument le champ *state*), afin d'ajouter un nouvel enregistrement dans chaque fichier texte contenant les résultats du calcul numérique.

Il convient de souligner un aspect important concernant le temps et qui est pris en compte dans notre travail. Vu que l'animation du mouvement du pendule ainsi que la représentation graphique des courbes associées doivent être faites en "temps réel", il faut qu'elles soient réactualisées à chaque signal émis par le *timer*. Par conséquent, la période du *timer* doit être suffisante pour l'accomplissement du calcul numérique proprement dit ainsi que pour la mise à jour des affichages graphiques correspondants.
Afin de s'assurer que cette condition est bien remplie, on a mesuré les temps nécessaires pour la réalisation des opérations mentionnées ci-dessus. Normalement, ces temps peuvent varier en fonction des différents paramètres qui définissent le problème concret à résoudre mais aussi en fonction des caractéristiques de l'ordinateur utilisé pour la simulation. Concrètement, on a appelé la méthode statique *nanoTime* de la classe standard *System* au début et à la fin d'une certaine tâche (par exemple avant et après l'appel de la méthode *solve* de la classe *Simulation*), ce qui nous a permis d'obtenir le temps d'exécution de cette tâche (en nanosecondes). Il faut quand même préciser que la valeur du temps ainsi obtenue peut être légèrement affectée par d'autres tâches de fond réalisées par le même cœur du processeur qui fait tourner la machine virtuelle. Cependant, c'est exactement ce temps qui nous intéresse car c'est lui qui doit permettre l'avancement de la simulation en "temps réel".
Plusieurs cas tests (benchmark tests) ont été effectués sur des ordinateurs avec des configurations différentes. Chaque test a été d'abord exécuté un nombre suffisant de fois, afin de permettre à la machine virtuelle de dépasser la période "d'échauffement de code" et d'arriver à un état d'équilibre caractérisé par un mode mixte d'exécution. Plus précisément, après le "code warmup", la machine virtuelle optimise son exécution grâce à son compilateur *Just-In-Time* (qui compile certaines méthodes englobant des blocs de code très souvent utilisés directement en code natif). Pour tous les essais ainsi menés, les temps d'exécution mesurés ont été suffisamment petits pour assurer l'avancement des simulations correspondantes en "temps réel".
Par souci de rigueur, on a refait les mêmes tests en utilisant aussi la méthode *getCurrentThreadCpuTime* de la classe standard *ThreadMXBean* du package *java.lang.management*. Cette méthode ressemble à la méthode *nanoTime* mais, suite à son appel, le temps courant (le "timestamp") retourné en nanosecondes correspond cette fois à un thread particulier (et non plus au CPU global). À nouveau, les temps mesurés ont confirmé que les simulations correspondantes avançaient en "temps réel".

En outre, grâce au bouton *Step* de l'interface graphique, l'utilisateur peut faire avancer la simulation "pas à pas". Plus précisément, chaque fois que

l'utilisateur clique sur le bouton *Step*, le gestionnaire d'événements qui lui est associé est exécuté automatiquement et appelle la méthode (sans arguments) *updateGUI_Step* de la classe *MainPanel*. Cette méthode appelle, à son tour, la méthode *updateGUI* de la même classe, en lui passant en argument la valeur du pas de temps avec lequel la simulation doit avancer, exprimé en millisecondes. Comme on l'a déjà vu, la valeur de ce pas est précisée par l'intermédiaire du champ static et final *FRAMES_PER_SECOND* de la classe *MainPanel* (qui vaut par défaut 50 cadres par seconde et donne ainsi un pas de temps de 20 millisecondes).

Quant à la distance parcourue par le chariot, elle est également mise à l'échelle pour la représentation animée du mouvement du système formé par la chariot et le pendule (mais pas pour la représentation graphique des courbes). En fait, le facteur d'échelle pour les longueurs est défini par le champ static et final *SCALE* de la classe *AnimationPanel* (et il a la valeur par défaut de 100, ce qui revient à représenter un centimètre du "monde réel" par un pixel à l'écran). Par contre, l'angle *theta* correspondant à la déviation de la tige par rapport à la verticale est représenté en grandeur réelle.
Il convient de mentionner que le système d'axes de coordonnées utilisé par défaut pour les représentations graphiques bidimensionnelles en Java place l'origine au coin supérieur gauche du container utilisé pour le dessin et considère l'axe horizontal orienté positivement vers la droite et l'axe vertical orienté positivement vers le bas. Par conséquent, afin de représenter correctement notre animation, il a fallu changer l'orientation de l'axe vertical (dans la méthode *paintAnimation* de la classe *AnimationPanel*, à l'aide la méthode *scale* de la classe prédéfinie *Graphics2D*).

Chapitre 4

Conclusions

4.1 Synthèse

Dans le cadre de ce travail de Bachelor, j'ai réalisé un simulateur d'un pendule inversé. Le projet est parti de zéro (from scratch) et j'ai investi un temps assez conséquent à la fois dans la conception de l'approche et dans son implémentation informatique à l'aide du langage Java et en utilisant l'environnement de programmation Eclipse, voir [13]. L'amélioration de mes connaissances durant mon travail a largement justifiée l'effort important consacré à sa réalisation.

4.2 Éléments informatiques utilisés

Pour réaliser ce projet, j'ai été amenée à me servir de :

- **Linux** (Mint *Sarah*), système d'exploitation de la famille UNIX, pour la gestion de toutes les autres ressources informatiques ;
- **Java** (version 8), langage de programmation orienté objets, pour l'écriture du code source ;
- **Eclipse** (version neon.1), environnement de programmation intégré, pour le développement du projet ;
- **Dropbox** (version 17.3.28), service de stockage et de partage, pour la sauvegarde des fichiers et le contrôle de versions ;
- **TeXstudio** (version 2.10.8), environnement intégré pour la création de documents LaTeX, pour la rédaction de ce rapport ;
- **GeoGebra** (version 4.0.34.0), logiciel de géométrie dynamique, pour la réalisation de tous les schémas ("home made") de ce rapport ;

- **Gimp** (version 2.8.16), éditeur d'images, pour le recadrage des schémas et dessins ;
- **Doxygen** (version 1.8.11), générateur de documentation, pour la réalisation de l'API du projet.

4.3 Points forts

Il convient de remarquer que ce projet a été construit à partir de zéro et de sorte à ce qu'il puisse être étendu dans la continuité. Dans ce sens, sa modularité longtemps réfléchie assure, d'une part, une séparation claire entre les classes qui réalisent le calcul numérique des grandeurs qui définissent le mouvement du système physique chariot-pendule-tige (et qui sont regroupées dans le package *alina.sim*) et les classes qui réalisent l'interfaçage avec l'utilisateur, y compris l'animation et les représentations graphiques des résultats (et ces dernières classes sont regroupées dans le package *alina.sim.ui*). D'autre part, le découpage du projet permettra la prise en compte d'éventuels aspects supplémentaires (physiques ou numériques) sans changements notables du code.

De plus, grâce à l'implémentation du design pattern *Strategy* dans le package *alina.sim.strategy*, de nouvelles méthodes de stabilisation du pendule inversé peuvent être ajoutées à volonté et localement, sans que l'ensemble du projet ou le code déjà écrit soient affectés.

Quant à l'utilisateur final, il tire profit de l'attention spéciale accordée à l'ergonomie, en général, et aux détails visuels, en particulier, et dispose d'une interface graphique intuitive et réactive qui lui assure une prise en main rapide et une exploitation conviviale du simulateur.

4.4 Développements futurs

Sans aucune fausse modestie, je pense que les objectifs fixés dans le cadre de ce projet ont bien été réalisés. Cependant, je suis consciente que des améliorations peuvent encore être apportées et j'en mentionne ci-dessous quelques exemples :

- la possibilité de préciser des conditions initiales supplémentaires (par exemple pour la vitesse angulaire $\dot{\theta}$) ;
- l'ajout d'autres stratégies de stabilisation ;
- l'utilisation de schémas numériques autres que le schéma d'Euler progressif ;

- la mise en ligne du projet (par exemple sous la forme d'une applet) afin de permettre son utilisation à distance ;
- une approche 3D.

4.5 Remerciements

Maintenant, à la fin de ce travail de Bachelor, je peux dire que j'ai eu beaucoup de plaisir à travailler dessus. Même lorsque j'ai rencontré des difficultés, je les ai regardées comme un défi et j'ai réussi à les surmonter.

Dans ce contexte, j'aimerais remercier tout d'abord le superviseur de ce projet, Monsieur le Professeur Ulrich Ultes-Nitsche, dont la compétence, la disponibilité et la gentillesse m'ont beaucoup impressionnée et aidée tout au long de mon travail.

En outre, je suis reconnaissante vis-à-vis de ma famille (en particulier de ma mère et de ma sœur) et de mes amis pour leurs soutient et encouragements. Aussi, je sais bien gré à mon père et à Monsieur Ismaïl Senhaji, pour la lecture de ce projet et pour les diverses idées proposées quant à une quelconque amélioration.

Annexe A

Documentation de l'implémentation (API) du projet

API - Simulateur du pendule inversé

Version 3.0

Généré par Doxygen 1.8.11

Table des matières

1 Index hiérarchique **1**

1.1 Hiérarchie des classes 1

2 Index des classes **3**

2.1 Liste des classes 3

3 Documentation des classes **5**

3.1 Référence de la classe alina.sim.ui.AnimationPanel 5

3.1.1 Description détaillée 5

3.1.2 Documentation des constructeurs et destructeur 6

3.1.2.1 AnimationPanel() 6

3.1.3 Documentation des fonctions membres 6

3.1.3.1 paintAnimation(Graphics2D g) 6

3.1.3.2 paintComponent(Graphics g) 6

3.1.3.3 update(Simulation.State state) 6

3.1.4 Documentation des données membres 6

3.1.4.1 CART_HEIGHT 6

3.1.4.2 CART_WIDTH 6

3.1.4.3 SCALE 7

3.2 Référence de la classe alina.sim.ui.ControlPanel 7

3.2.1 Description détaillée 7

3.2.2 Documentation des constructeurs et destructeur 7

3.2.2.1 ControlPanel(MainPanel mainPanel) 7

3.2.3 Documentation des fonctions membres 8

3.2.3.1 formatValue(int indexSlider, double value) . . . 8
3.2.3.2 getPrecision() . . . 8
3.2.3.3 getState() . . . 8
3.2.3.4 getValue(int indexSlider) . . . 8
3.2.3.5 initButtons() . . . 9
3.2.3.6 initSliders() . . . 9
3.2.3.7 pause() . . . 9
3.2.3.8 play() . . . 9
3.2.3.9 setPrecision(int precision) . . . 9
3.2.3.10 transformValue(int indexSlider, int value) . . . 9
3.2.3.11 updateLabel(int slider) . . . 9
3.3 Référence de la classe alina.sim.Main . . . 10
3.3.1 Description détaillée . . . 10
3.3.2 Documentation des fonctions membres . . . 10
3.3.2.1 main(String[] args) . . . 10
3.3.2.2 startInterface() . . . 10
3.4 Référence de la classe alina.sim.ui.MainPanel . . . 11
3.4.1 Description détaillée . . . 11
3.4.2 Documentation des constructeurs et destructeur . . . 11
3.4.2.1 MainPanel(Strategy[] strategies) . . . 11
3.4.3 Documentation des fonctions membres . . . 12
3.4.3.1 initInterface() . . . 12
3.4.3.2 isRunning() . . . 12
3.4.3.3 isStarted() . . . 12
3.4.3.4 pause() . . . 12
3.4.3.5 play() . . . 12
3.4.3.6 reset() . . . 12
3.4.3.7 setSpeed(double speed) . . . 12
3.4.3.8 setStrategy(int i) . . . 12
3.4.3.9 updateGUI(long nb_ms) . . . 12

3.4.3.10 updateGUI_Step() . . . 13
3.4.4 Documentation des données membres . . . 13
3.4.4.1 animationPanel . . . 13
3.4.4.2 controlPanel . . . 13
3.4.4.3 FRAMES_PER_SECOND . . . 13
3.4.4.4 plotPanel . . . 13
3.4.4.5 running . . . 13
3.4.4.6 sidePanel . . . 13
3.4.4.7 simulation . . . 13
3.4.4.8 speed . . . 14
3.4.4.9 strategies . . . 14
3.4.4.10 strategyBox . . . 14
3.4.4.11 strategyPanel . . . 14
3.4.4.12 timeLastFrame . . . 14
3.4.4.13 timer . . . 14
3.5 Référence de la classe alina.sim.strategy.ManualStrategy . . . 14
3.5.1 Description détaillée . . . 15
3.5.2 Documentation des constructeurs et destructeur . . . 15
3.5.2.1 ManualStrategy() . . . 15
3.5.3 Documentation des fonctions membres . . . 15
3.5.3.1 getPanel() . . . 15
3.5.3.2 initPanel() . . . 15
3.5.3.3 react(Simulation.State state) . . . 15
3.5.3.4 reset() . . . 16
3.6 Référence de la classe alina.sim.strategy.NullStrategy . . . 16
3.6.1 Description détaillée . . . 16
3.6.2 Documentation des constructeurs et destructeur . . . 16
3.6.2.1 NullStrategy() . . . 16
3.6.3 Documentation des fonctions membres . . . 17
3.6.3.1 getPanel() . . . 17

3.6.3.2 react(Simulation.State state) . . . 17
3.6.3.3 reset() . . . 17
3.7 Référence de la classe alina.sim.OutputFiles . . . 17
3.7.1 Description détaillée . . . 18
3.7.2 Documentation des constructeurs et destructeur . . . 18
3.7.2.1 OutputFiles() . . . 18
3.7.3 Documentation des fonctions membres . . . 18
3.7.3.1 close() . . . 18
3.7.3.2 reset() . . . 18
3.7.3.3 update(Simulation.State state) . . . 18
3.7.4 Documentation des données membres . . . 18
3.7.4.1 fileNames . . . 19
3.7.4.2 pwList . . . 19
3.7.4.3 SIZE . . . 19
3.8 Référence de la classe alina.sim.ui.PlotFrameAng . . . 19
3.8.1 Description détaillée . . . 19
3.8.2 Documentation des constructeurs et destructeur . . . 19
3.8.2.1 PlotFrameAng() . . . 19
3.8.3 Documentation des fonctions membres . . . 20
3.8.3.1 initInterface() . . . 20
3.8.3.2 reset() . . . 20
3.8.3.3 update(Simulation.State state) . . . 20
3.9 Référence de la classe alina.sim.ui.PlotPanel . . . 20
3.9.1 Description détaillée . . . 20
3.9.2 Documentation des constructeurs et destructeur . . . 21
3.9.2.1 PlotPanel() . . . 21
3.9.3 Documentation des fonctions membres . . . 21
3.9.3.1 initInterface() . . . 21
3.9.3.2 reset() . . . 21
3.9.3.3 update(Simulation.State state) . . . 21

3.10 Référence de la classe alina.sim.ui.SidePanel . . . 21
3.10.1 Description détaillée . . . 22
3.10.2 Documentation des constructeurs et destructeur . . . 22
3.10.2.1 SidePanel() . . . 22
3.10.3 Documentation des fonctions membres . . . 22
3.10.3.1 addPanel(JPanel panel, String title) . . . 22
3.10.3.2 createBorder(String title) . . . 22
3.10.3.3 removePanel(JPanel panel) . . . 22
3.11 Référence de la classe alina.sim.strategy.SimpleStrategy . . . 22
3.11.1 Description détaillée . . . 23
3.11.2 Documentation des constructeurs et destructeur . . . 23
3.11.2.1 SimpleStrategy() . . . 23
3.11.3 Documentation des fonctions membres . . . 23
3.11.3.1 getPanel() . . . 23
3.11.3.2 initPanel() . . . 23
3.11.3.3 react(Simulation.State state) . . . 23
3.11.3.4 reset() . . . 24
3.12 Référence de la classe alina.sim.Simulation . . . 24
3.12.1 Description détaillée . . . 25
3.12.2 Documentation des constructeurs et destructeur . . . 25
3.12.2.1 Simulation() . . . 25
3.12.2.2 Simulation(State startState, Strategy strategy, int precision) . . . 25
3.12.3 Documentation des fonctions membres . . . 25
3.12.3.1 getState() . . . 25
3.12.3.2 getStrategy() . . . 26
3.12.3.3 isStarted() . . . 26
3.12.3.4 reset() . . . 26
3.12.3.5 reset(State state) . . . 26
3.12.3.6 setPrecision(int precision) . . . 26
3.12.3.7 setStrategy(Strategy strategy) . . . 26

3.12.3.8 solve(long nb_ms) . . . 27
3.12.3.9 solveStep(double dt) . . . 27
3.12.4 Documentation des données membres . . . 27
3.12.4.1 g . . . 27
3.12.4.2 precision . . . 27
3.12.4.3 startState . . . 27
3.12.4.4 state . . . 27
3.12.4.5 strategy . . . 27
3.13 Référence de la classe alina.sim.Simulation.State . . . 28
3.13.1 Description détaillée . . . 28
3.13.2 Documentation des constructeurs et destructeur . . . 28
3.13.2.1 State() . . . 28
3.13.2.2 State(State state) . . . 28
3.13.3 Documentation des données membres . . . 28
3.13.3.1 aAng . . . 28
3.13.3.2 ax . . . 29
3.13.3.3 ay . . . 29
3.13.3.4 failed . . . 29
3.13.3.5 I . . . 29
3.13.3.6 m . . . 29
3.13.3.7 M . . . 29
3.13.3.8 started . . . 29
3.13.3.9 t . . . 29
3.13.3.10 theta . . . 29
3.13.3.11 vAng . . . 29
3.13.3.12 vx . . . 30
3.13.3.13 vy . . . 30
3.13.3.14 x . . . 30
3.13.3.15 y . . . 30
3.14 Référence de l'interface alina.sim.strategy.Strategy . . . 30
3.14.1 Description détaillée . . . 30
3.14.2 Documentation des fonctions membres . . . 31
3.14.2.1 getPanel() . . . 31
3.14.2.2 react(Simulation.State state) . . . 31
3.14.2.3 reset() . . . 31
3.15 Référence de la classe alina.sim.ui.StrategyPanel . . . 31
3.15.1 Description détaillée . . . 32
3.15.2 Documentation des constructeurs et destructeur . . . 32
3.15.2.1 StrategyPanel(Strategy[] strategies) . . . 32
3.15.3 Documentation des fonctions membres . . . 32
3.15.3.1 show(int index) . . . 32

Chapitre 1

Index hiérarchique

1.1 Hiérarchie des classes

Cette liste d'héritage est classée approximativement par ordre alphabétique :

alina.sim.ui.AnimationPanel 5
alina.sim.ui.ControlPanel 7
alina.sim.Main 10
alina.sim.ui.MainPanel 11
alina.sim.OutputFiles 17
alina.sim.ui.PlotFrameAng 19
alina.sim.ui.PlotPanel 20
alina.sim.ui.SidePanel 21
alina.sim.Simulation 24
alina.sim.Simulation.State 28
alina.sim.strategy.Strategy 30
- alina.sim.strategy.ManualStrategy 14
- alina.sim.strategy.NullStrategy 16
- alina.sim.strategy.SimpleStrategy 22

alina.sim.ui.StrategyPanel 31

Chapitre 2

Index des classes

2.1 Liste des classes

Liste des classes, structures, unions et interfaces avec une brève description :

alina.sim.ui.AnimationPanel 5
alina.sim.ui.ControlPanel 7
alina.sim.Main 10
alina.sim.ui.MainPanel 11
alina.sim.strategy.ManualStrategy 14
alina.sim.strategy.NullStrategy 16
alina.sim.OutputFiles 17
alina.sim.ui.PlotFrameAng 19
alina.sim.ui.PlotPanel 20
alina.sim.ui.SidePanel 21
alina.sim.strategy.SimpleStrategy 22
alina.sim.Simulation 24
alina.sim.Simulation.State 28
alina.sim.strategy.Strategy 30
alina.sim.ui.StrategyPanel 31

Chapitre 3

Documentation des classes

3.1 Référence de la classe alina.sim.ui.AnimationPanel

Est dérivée de JPanel.

Fonctions membres publiques

- AnimationPanel ()
- void update (Simulation.State state)

Attributs publics statiques

- static final int SCALE = 100
- static final double CART_WIDTH = 0.4
- static final double CART_HEIGHT = 0.2

Fonctions membres protégées

- void paintComponent (Graphics g)

Fonctions membres privées

- void paintAnimation (Graphics2D g)

3.1.1 Description détaillée

Cette classe dessine l'état du système, la simulation (chariot, tige, boule...) ainsi que les forces associées.

Auteur

alina petrescu

Version

2.0

3.1.2 Documentation des constructeurs et destructeur

3.1.2.1 alina.sim.ui.AnimationPanel.AnimationPanel ( )

Constructeur public sans arguments qui donne les dimensions par défaut au panneau de l'animation.

3.1.3 Documentation des fonctions membres

3.1.3.1 void alina.sim.ui.AnimationPanel.paintAnimation (Graphics2D *g*) [private]

Cette méthode met tout d'abord les paramètre de l'animation à l'échelle de la simulation. Elle place ensuite le pendule inverse au milieu du panneau et y associe ses forces avec différentes couleurs. Cette méthode permet aussi de "suivre" le pendule s'il sort du panneau.

Paramètres

g	le "pinceau" avec lequel on dessine l'animation

3.1.3.2 void alina.sim.ui.AnimationPanel.paintComponent (Graphics *g*) [protected]

Cette méthode dessine les composantes de l'animation.

Paramètres

g	le "pinceau" avec lequel on dessine l'animation

3.1.3.3 void alina.sim.ui.AnimationPanel.update (Simulation.State *state*)

Cette méthode affiche l'état du système à chaque prochaine itération.

Paramètres

state	l'état courant de la simulaton

3.1.4 Documentation des données membres

3.1.4.1 final double alina.sim.ui.AnimationPanel.CART_HEIGHT = 0.2 [static]

La hauteur initiale du chariot.

3.1.4.2 final double alina.sim.ui.AnimationPanel.CART_WIDTH = 0.4 [static]

La largeur initiale du chariot.

3.1.4.3 final int alina.sim.ui.AnimationPanel.SCALE = 100 [static]

L'échelle des longueurs, i.e. le nombre de pixels utilisés pour représenter une longueur réelle d'un mètre.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/ui/AnimationPanel.java

3.2 Référence de la classe alina.sim.ui.ControlPanel

Est dérivée de JPanel.

Fonctions membres publiques

— ControlPanel (MainPanel mainPanel)
— void play ()
— void pause ()
— int getPrecision ()
— void setPrecision (int precision)
— double getValue (int indexSlider)
— Simulation.State getState ()

Fonctions membres privées

— void initSliders ()
— void initButtons ()
— void updateLabel (int slider)
— double transformValue (int indexSlider, int value)
— String formatValue (int indexSlider, double value)

3.2.1 Description détaillée

Cette classe permet de démarrer ou de mettre en pause la simulation, ainsi que de l'accélérer ou de la ralentir. Elle définit entre autre les paramètres de départ du système (angle de départ, masse du chariot, masse du pendule, etc.).

Auteur

alina petrescu

Version

2.0

3.2.2 Documentation des constructeurs et destructeur

3.2.2.1 alina.sim.ui.ControlPanel.ControlPanel (MainPanel *mainPanel*)

Constructeur public avec un argument qui initialise le panneau de contrôle.

Paramètres

mainPanel	le panneau de la fenêtre principale

3.2.3 Documentation des fonctions membres

3.2.3.1 String alina.sim.ui.ControlPanel.formatValue (int *indexSlider,* double *value*) [private]

Cette méthode met un format spécifique à chaque curseur.

Paramètres

indexSlider	l'indice du curseur courant
value	la valeur du curseur courant

Renvoie

le nouveau format de chaque curseur

3.2.3.2 int alina.sim.ui.ControlPanel.getPrecision ( )

Cette méthode retourne la précision.

3.2.3.3 Simulation.State alina.sim.ui.ControlPanel.getState ( )

Cette méthode permet de retourner les valeurs "normales" des curseurs.

Renvoie

l'état courant du système

3.2.3.4 double alina.sim.ui.ControlPanel.getValue (int *indexSlider*)

Retourne la valeur courante transformée du curseur courant.

Paramètres

indexSlider	l'indice du curseur courant

Renvoie

la valeur transformée du "curseur argument"

3.2.3.5 void alina.sim.ui.ControlPanel.initButtons ( ) [private]

Cette méthode initialise tous les boutons. Elle leur met les étiquettes et leur ajoute les écouteurs et les événements associés à chacun des boutons.

3.2.3.6 void alina.sim.ui.ControlPanel.initSliders ( ) [private]

Cette méthode initalise tous les sliders. Elle mets les étiquettes, les valeurs des curseurs, ainsi que l'espacement entre chaque "tick" des curseurs. De plus, elle leur ajoutent les écouteurs ainsi que les événements associés à chacun des sliders.

3.2.3.7 void alina.sim.ui.ControlPanel.pause ( )

Cette méthode affiche PLAY (à la place de PAUSE) lorsque le bouton PAUSE a été appuyé pour mettre en pause la simulation. De plus, cette méthode réactive tous les sliders.

3.2.3.8 void alina.sim.ui.ControlPanel.play ( )

Cette méthode affiche PAUSE (à la place de PLAY) lorsque le bouton PLAY a été appuyé pour démarrer la simulation. De plus, cette méthode "désactive" (met en gris) tous les sliders sauf celui du SPEED.

3.2.3.9 void alina.sim.ui.ControlPanel.setPrecision (int *precision*)

Cette méthode met en place la valeur de la précision.

Paramètres

precision	la précision de la simulation

3.2.3.10 double alina.sim.ui.ControlPanel.transformValue (int *indexSlider,* int *value*) [private]

Cette méthode transforme la valeur d'un curseur en valeur "réelle" pour la simulation.

Paramètres

indexSlider	l'indice du curseur courant
value	la valeur du curseur courant

Renvoie

la valeur "réelle"

3.2.3.11 void alina.sim.ui.ControlPanel.updateLabel (int *slider*) [private]

Cette méthode permet de mettre à jour la valeur du curseur lorsque celui-ci est bougé.

Paramètres

slider	le curseur "courant"

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/ui/ControlPanel.java

3.3 Référence de la classe alina.sim.Main

Fonctions membres publiques statiques

— static void startInterface ()
— static void main (String[ ] args)

3.3.1 Description détaillée

Cette classe est la classe principale avec laquelle démarre le projet. A l'exécution de la méthode main(), le système de fenêtres imbriquées est créé et l'utilisateur peut commencer à interagir avec l'interface graphique.

Auteur

alina petrescu

Version

2.0

3.3.2 Documentation des fonctions membres

3.3.2.1 static void alina.sim.Main.main (String[ ] *args*) [static]

Cette méthode est le point d'entrée du projet. Elle est la première méthode à être exécutée par la JVM.

Paramètres

args	tableau de strings (qui peut stocker les paramètres donnés sur la ligne de commande)

3.3.2.2 static void alina.sim.Main.startInterface ( ) [static]

Cette méthode définit l'état initial de l'interface graphique.

La documentation de cette classe a été générée à partir du fichier suivant :

— src/alina/sim/Main.java

3.4 Référence de la classe alina.sim.ui.MainPanel

Est dérivée de JPanel.

Fonctions membres publiques

- MainPanel (Strategy[ ] strategies)
- void play ()
- void pause ()
- void reset ()
- void updateGUI (long nb_ms)
- void updateGUI_Step ()
- void setSpeed (double speed)
- void setStrategy (int i)
- boolean isRunning ()
- boolean isStarted ()

Attributs publics statiques

- static final int FRAMES_PER_SECOND = 50

Fonctions membres privées

- void initInterface ()

Attributs privés

- final Simulation simulation
- final Strategy[ ] strategies
- final SidePanel sidePanel
- final AnimationPanel animationPanel
- final PlotPanel plotPanel
- final ControlPanel controlPanel
- final JComboBox< String > strategyBox
- final StrategyPanel strategyPanel
- final Timer timer
- double speed = 1
- long timeLastFrame
- boolean running

3.4.1 Description détaillée

Cette classe représente le contenu de la fenêtre principale et correspond à un panneau swing qui regroupe les autres containers intermédiaires et objets grapiques atomiques (widgets).

Auteur

alina petrescu

Version

2.0

3.4.2 Documentation des constructeurs et destructeur

3.4.2.1 alina.sim.ui.MainPanel.MainPanel (Strategy[ ] *strategies*)

Ce contructeur public initialise les panneaux et permet de choisir la stratégie voulue.

Paramètres

strategies	tableau de toutes les stratégies disponibles

3.4.3 Documentation des fonctions membres

3.4.3.1 void alina.sim.ui.MainPanel.initInterface ( ) [private]

Cette méthode place tous la panneaux d'affichage et initialise toute l'interface graphique. De plus, elle ajoute l'écouteur correpondant à la stratégie sélectionnée.

3.4.3.2 boolean alina.sim.ui.MainPanel.isRunning ( )

Cette méthode permet de savoir si la simulation est en marche ou pas.

3.4.3.3 boolean alina.sim.ui.MainPanel.isStarted ( )

Cette méthode permet de savoir si la simulation est lancée ou pas.

3.4.3.4 void alina.sim.ui.MainPanel.pause ( )

Cette méthode permet de mettre en pause la simulation.

3.4.3.5 void alina.sim.ui.MainPanel.play ( )

Cette méthode permet de lancer la simulation.

3.4.3.6 void alina.sim.ui.MainPanel.reset ( )

Cette méthode permet de réinitialiser les paramètres de la simulation.

3.4.3.7 void alina.sim.ui.MainPanel.setSpeed (double *speed*)

Cette méthode permet de changer la vitesse de déroulement de la simulation.

3.4.3.8 void alina.sim.ui.MainPanel.setStrategy (int *i*)

Cette méthode permet de choisir la stratégie voulue.

3.4.3.9 void alina.sim.ui.MainPanel.updateGUI (long *nb_ms*)

Cette méthode permet de mettre à jour l'interface graphique (à savoir, l'animation et les graphiques), après l'avancement de la simulation durant un intervalle de temps donné en paramètre.

Paramètres

nb_ms	nombre de millisecondes d'attente

3.4.3.10 void alina.sim.ui.MainPanel.updateGUI_Step ( )

Cette méthode permet de mettre à jour l'interface graphique après l'avancement de la simulation durant un intervalle de temps (très petit), à savoir le laps (ou le pas) de temps écoulé entre deux cadres successifs.

3.4.4 Documentation des données membres

3.4.4.1 final AnimationPanel alina.sim.ui.MainPanel.animationPanel [private]

Container intermédiaire qui affiche l'animation (chariot + pendule).

3.4.4.2 final ControlPanel alina.sim.ui.MainPanel.controlPanel [private]

Container intermédiaire qui regroupe

3.4.4.3 final int alina.sim.ui.MainPanel.FRAMES_PER_SECOND = 50 [static]

La fréquence de la simulation (en nombre de cadres par seconde).

3.4.4.4 final PlotPanel alina.sim.ui.MainPanel.plotPanel [private]

Container intermédiaire qui affiche les courbes graphiques.

3.4.4.5 boolean alina.sim.ui.MainPanel.running [private]

Marqueur booléen qui permet de savoir si la simulation est en marche ou pas. S'il a la valeur false (suite à un clic sur le bouton PAUSE, les événements ActionEvent émis par le timer restent sans effets.

3.4.4.6 final SidePanel alina.sim.ui.MainPanel.sidePanel [private]

Container intermédiaire qui regroupe

3.4.4.7 final Simulation alina.sim.ui.MainPanel.simulation [private]

Regroupement des informations pertinentes concernant la simulation.

3.4.4.8 double alina.sim.ui.MainPanel.speed = 1 [private]

La vitesse de déroulement de la simulation. (speed = 1 correspond au déroulement en temps réel)

3.4.4.9 final Strategy [ ] alina.sim.ui.MainPanel.strategies [private]

Tableau avec les stratégies pour la stabilisation du pendule.

3.4.4.10 final JComboBox<String> alina.sim.ui.MainPanel.strategyBox [private]

Combo box qui permet de sélectionner la stratégie voulue.

3.4.4.11 final StrategyPanel alina.sim.ui.MainPanel.strategyPanel [private]

Container intermédiaire qui permet de paramétrer la stratégie choisie.

3.4.4.12 long alina.sim.ui.MainPanel.timeLastFrame [private]

La valeur du temps (le timestamp) correspondant à l'émission du dernier événement actionEvent lancé par la timer pendant le déroulement de la simulation (avec le champ running ayant la valeur true).

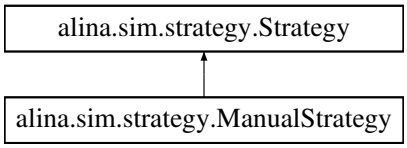
3.4.4.13 final Timer alina.sim.ui.MainPanel.timer [private]

Le moteur de l'animation.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/ui/MainPanel.java

3.5 Référence de la classe alina.sim.strategy.ManualStrategy

Graphe d'héritage de alina.sim.strategy.ManualStrategy :



Fonctions membres publiques

— ManualStrategy ()
— void reset ()
— double react (Simulation.State state)
— JPanel getPanel ()

Fonctions membres privées

— void initPanel ()

3.5.1 Description détaillée

Cette classe implémente une stratégie de test où on peut diriger le chariot manuellement avec un slider ad-hoc.

Auteur

alina

Version

1.0

3.5.2 Documentation des constructeurs et destructeur

3.5.2.1 alina.sim.strategy.ManualStrategy.ManualStrategy ( )

Le constructeur SANS arguments de la classe ManualStrategy.

3.5.3 Documentation des fonctions membres

3.5.3.1 JPanel alina.sim.strategy.ManualStrategy.getPanel ( )

Cette méthode retourne le panneau de paramétrage de la stratégie.

Renvoie

le panneau de paramétrage

Implémente alina.sim.strategy.Strategy.

3.5.3.2 void alina.sim.strategy.ManualStrategy.initPanel ( ) [private]

Cette méthode initialise le panneau de paramétrage de la stratégie courante.

3.5.3.3 double alina.sim.strategy.ManualStrategy.react (Simulation.State *state*)

Cette méthode calcule la force à appliquer par le "moteur" du chariot pour tenter de garder le pendule inversé en équilibre.

Paramètres

state	l'état actuel du système physique chariot-tige-pendule

Renvoie

la force à appliquer sur le chariot

Implémente alina.sim.strategy.Strategy.

3.5.3.4 void alina.sim.strategy.ManualStrategy.reset ( )

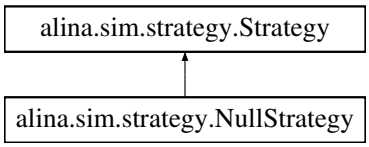
Cette méthode a une implémentation nulle pour cette stratégie.

Implémente alina.sim.strategy.Strategy.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/strategy/ManualStrategy.java

3.6 Référence de la classe alina.sim.strategy.NullStrategy

Graphe d'héritage de alina.sim.strategy.NullStrategy :



Fonctions membres publiques

— NullStrategy ()
— void reset ()
— double react (Simulation.State state)
— JPanel getPanel ()

3.6.1 Description détaillée

Cette classe est prévue pour faire des tests avec des nouvelles stratégies avant leur implémentation dans des classes spécifiques.

Auteur

alina

Version

2.0

3.6.2 Documentation des constructeurs et destructeur

3.6.2.1 alina.sim.strategy.NullStrategy.NullStrategy ( )

Le constructeur SANS arguments de la classe NullStrategy.

3.6.3 Documentation des fonctions membres

3.6.3.1 JPanel alina.sim.strategy.NullStrategy.getPanel ( )

Cette méthode retourne le panneau de paramétrage de la stratégie.

Renvoie

le panneau de paramétrage

Implémente alina.sim.strategy.Strategy.

3.6.3.2 double alina.sim.strategy.NullStrategy.react (Simulation.State *state*)

Cette méthode calcule la force à appliquer par le "moteur" du chariot pour tenter de garder le pendule inversé en équilibre.

Paramètres

state	l'état actuel du système physique chariot-tige-pendule

Renvoie

la force à appliquer sur le chariot

Implémente alina.sim.strategy.Strategy.

3.6.3.3 void alina.sim.strategy.NullStrategy.reset ( )

Cette méthode a une implémentation nulle pour cette stratégie.

Implémente alina.sim.strategy.Strategy.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/strategy/NullStrategy.java

3.7 Référence de la classe alina.sim.OutputFiles

Fonctions membres publiques

— OutputFiles ()
— void update (Simulation.State state)
— void reset ()
— void close ()

Attributs privés

— final int SIZE = 4
— List< PrintWriter > pwList = new ArrayList<>(SIZE)
— String[ ] fileNames = {"xFile", "vxFile", "axFile", "thetaFile"}

3.7.1 Description détaillée

Cette classe permet de stocker dans des fichiers texte les résultats du calcul numérique obtenus durant la simulation, à savoir : la position, la vitesse et l'accélération du chariot, ainsi que l'angle theta pour chaque moment courant.

Auteur

alina petrescu

Version

1.0

3.7.2 Documentation des constructeurs et destructeur

3.7.2.1 alina.sim.OutputFiles.OutputFiles ( )

Constructeur public SANS arguments qui associe à chaque fichier texte à créer un objet de type PrintWriter qui permet au programme d'y stocker les données voulues.

3.7.3 Documentation des fonctions membres

3.7.3.1 void alina.sim.OutputFiles.close ( )

Cette méthode permet de fermer les canaux de communication entre le programme et les fichiers texte ouverts en écriture.

3.7.3.2 void alina.sim.OutputFiles.reset ( )

Cette méthode permet de remplacer les fichiers texte ouverts en écriture par des nouveaux fichiers vides et dans lesquels l'écriture commence à nouveau tout au début.

3.7.3.3 void alina.sim.OutputFiles.update (Simulation.State *state*)

Cette méthode écrit dans chaque fichier texte une nouvelle ligne contenant la valeur correspondante de l'état courant de la simulation et le temps représentant le moment de l'appel de la méthode.

Paramètres

state	l'état courant de la simulation

3.7.4 Documentation des données membres

3.7.4.1 String [ ] alina.sim.OutputFiles.fileNames = {"xFile", "vxFile", "axFile", "thetaFile"} [private]

Le tableau avec les noms des fichiers texte.

3.7.4.2 List<PrintWriter> alina.sim.OutputFiles.pwList = new ArrayList<>(SIZE) [private]

La liste d'objets PrintWriter à connecter aux fichiers texte.

3.7.4.3 final int alina.sim.OutputFiles.SIZE = 4 [private]

Le nombre de fichiers texte à créer.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/OutputFiles.java

3.8 Référence de la classe alina.sim.ui.PlotFrameAng

Est dérivée de JFrame.

Fonctions membres publiques

— PlotFrameAng ()
— void reset ()
— void update (Simulation.State state)

Fonctions membres privées

— void initInterface ()

3.8.1 Description détaillée

Cette classe dessine le graphique et les courbes correspondantes à la simulation.

Auteur

alina petrescu

Version

2.0

3.8.2 Documentation des constructeurs et destructeur

3.8.2.1 alina.sim.ui.PlotFrameAng.PlotFrameAng ( )

Cette méthode (constructeur public) initialise le panneau de dessin.

3.8.3 Documentation des fonctions membres

3.8.3.1 void alina.sim.ui.PlotFrameAng.initInterface ( ) [private]

Cette méthode permet de mettre les couleurs, les unités et les bonnes légandes des axes.

3.8.3.2 void alina.sim.ui.PlotFrameAng.reset ( )

Cette méthode permet de réinitialiser le panneau du graphique.

3.8.3.3 void alina.sim.ui.PlotFrameAng.update (Simulation.State *state*)

Cette méthode permet d'ajouter les points sur le graphique.

Paramètres

state	l'état courant de la simulation

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/ui/PlotFrameAng.java

3.9 Référence de la classe alina.sim.ui.PlotPanel

Est dérivée de JPanel.

Fonctions membres publiques

— PlotPanel ()
— void reset ()
— void update (Simulation.State state)

Fonctions membres privées

— void initInterface ()

3.9.1 Description détaillée

Cette classe dessine le graphique et les courbes correspondantes à la simulation.

Auteur

alina petrescu

Version

2.0

3.9.2 Documentation des constructeurs et destructeur

3.9.2.1 alina.sim.ui.PlotPanel.PlotPanel ( )

Constructeur public qui initialise le panneau de dessin.

3.9.3 Documentation des fonctions membres

3.9.3.1 void alina.sim.ui.PlotPanel.initInterface ( ) [private]

Cette méthode permet de mettre les couleurs, les unités et les bonnes légendes des axes.

3.9.3.2 void alina.sim.ui.PlotPanel.reset ( )

Cette méthode permet de réinitialiser le panneau du graphique.

3.9.3.3 void alina.sim.ui.PlotPanel.update (Simulation.State *state*)

Cette méthode permet d'ajouter les points sur le graphique.

Paramètres

state	l'état courant de la simulation

La documentation de cette classe a été générée à partir du fichier suivant :

— src/alina/sim/ui/PlotPanel.java

3.10 Référence de la classe alina.sim.ui.SidePanel

Est dérivée de JScrollPane.

Fonctions membres publiques

— SidePanel ()
— void addPanel (JPanel panel, String title)
— void removePanel (JPanel panel)

Fonctions membres privées statiques

— static Border createBorder (String title)

3.10.1 Description détaillée

Cette classe implémente le panneau à droite de la simulation. Elle contient le panneau de contrôle et affiche une barre de défilement verticale lorsque le panneau devient trop grand.

Auteur

alina petrescu

Version

2.0

3.10.2 Documentation des constructeurs et destructeur

3.10.2.1 alina.sim.ui.SidePanel.SidePanel ( )

Cette méthode (constructeur public) permet d'initialiser le panneau.

3.10.3 Documentation des fonctions membres

3.10.3.1 void alina.sim.ui.SidePanel.addPanel (JPanel *panel,* String *title*)

Cette méthode permet de d'ajouter le panneau et de lui ajouter un titre.

3.10.3.2 static Border alina.sim.ui.SidePanel.createBorder (String *title*) [static],[private]

Cette méthode permet de créer les bordures.

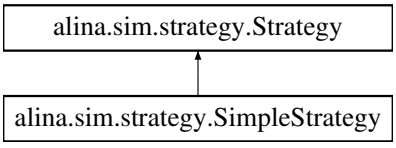
3.10.3.3 void alina.sim.ui.SidePanel.removePanel (JPanel *panel*)

Cette méthode permet de supprimer le panneau.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/ui/SidePanel.java

3.11 Référence de la classe alina.sim.strategy.SimpleStrategy

Graphe d'héritage de alina.sim.strategy.SimpleStrategy :



Fonctions membres publiques

- SimpleStrategy ()
- void reset ()
- double react (Simulation.State state)
- JPanel getPanel ()

Fonctions membres privées

- void initPanel ()

3.11.1 Description détaillée

Cette classe implémente la stratégie par défaut de la simulation. Cette stratégie est basée sur le "PID controller" (proportionnel-intégrateur-dérivateur) qui calcule l'erreur afin de la minimiser.

Auteur

alina

Version

1.0

3.11.2 Documentation des constructeurs et destructeur

3.11.2.1 alina.sim.strategy.SimpleStrategy.SimpleStrategy ( )

Le constructeur SANS arguments de la classe SimpleStrategy.

3.11.3 Documentation des fonctions membres

3.11.3.1 JPanel alina.sim.strategy.SimpleStrategy.getPanel ( )

Cette méthode retourne le panneau de paramétrage de la stratégie.

Renvoie

le panneau de paramétrage

Implémente alina.sim.strategy.Strategy.

3.11.3.2 void alina.sim.strategy.SimpleStrategy.initPanel ( ) [private]

Cette méthode initialise le panneau de paramétrage de la stratégie courante.

3.11.3.3 double alina.sim.strategy.SimpleStrategy.react (Simulation.State *state*)

Cette méthode calcule la force à appliquer par le "moteur" du chariot pour tenter de garder le pendule inversé en équilibre.

Paramètres

state	l'état actuel du système physique chariot-tige-pendule

Renvoie

la force à appliquer sur le chariot

Implémente alina.sim.strategy.Strategy.

3.11.3.4 void alina.sim.strategy.SimpleStrategy.reset ( )

Cette méthode a une implémentation nulle pour cette stratégie.

Implémente alina.sim.strategy.Strategy.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/strategy/SimpleStrategy.java

3.12 Référence de la classe alina.sim.Simulation

Classes

— class State

Fonctions membres publiques

— Simulation ()
— Simulation (State startState, Strategy strategy, int precision)
— void solveStep (double dt)
— void solve (long nb_ms)
— void reset ()
— void reset (State state)
— State getState ()
— void setPrecision (int precision)
— boolean isStarted ()
— Strategy getStrategy ()
— void setStrategy (Strategy strategy)

Attributs publics

— State startState
— State state
— Strategy strategy

Attributs publics statiques

— static final double g = 9.81

Attributs privés statiques

— static int precision = 1000

3.12.1 Description détaillée

Cette classe regroupe les informations pertinentes concernant la simulation. En outre, elle implémente les lois de la physique appliquées au système étudié. En fonction de l'état du système à un moment donné, elle permet d'obtenir l'état du système après un certain intervalle de temps.

Auteur

alina petrescu

Version

2.0

3.12.2 Documentation des constructeurs et destructeur

3.12.2.1 alina.sim.Simulation.Simulation ( )

Le constructeur SANS arguments de la classe simulation.

3.12.2.2 alina.sim.Simulation.Simulation (State *startState,* Strategy *strategy,* int *precision*)

Le constructeur AVEC arguments de la classe simulation.

Paramètres

startState	l'état de départ de la simulation
strategy	la stratégie choisie pour la simulation
precision	la précision de la simulation

3.12.3 Documentation des fonctions membres

3.12.3.1 State alina.sim.Simulation.getState ( )

Cette méthode retourne l'état courant du pendule.

Renvoie

l'état courant du pendule

3.12.3.2 Strategy alina.sim.Simulation.getStrategy ( )

Cette méthode retourne la stratégie utilisée par la simulation.

Renvoie

la stratégie utilisée par la simulation

3.12.3.3 boolean alina.sim.Simulation.isStarted ( )

Cette méthode permet de savoir si la simulation est en marche ou pas.

Renvoie

vrai si la simulation est en marche et faux autrement

3.12.3.4 void alina.sim.Simulation.reset ( )

Cette méthode remet l'état courant de la simulation à sa valeur de départ et réinitialise la stratégie.

3.12.3.5 void alina.sim.Simulation.reset (State *state*)

Cette méthode remet l'état de départ et l'état courant de la simulation à la valeur donnée comme argument.

Paramètres

state	cet état devient l'état de départ et l'état courant de la simulation

3.12.3.6 void alina.sim.Simulation.setPrecision (int *precision*)

Cette méthode permet de changer la précision du calcul numérique.

Paramètres

precision	la nouvelle précision du calcul numérique

3.12.3.7 void alina.sim.Simulation.setStrategy (Strategy *strategy*)

Cette méthode permet de changer la stratégie utilisée par la simulation.

Paramètres

strategy	la nouvelle stratégie

3.12.3.8 void alina.sim.Simulation.solve (long *nb_ms*)

Cette méthode utilise la méthode solveStep pour permettre de faire avancer la simulation durant un intervalle de temps donné en paramètre.

Paramètres

nb_ms	l'intervalle de temps (en ms)

3.12.3.9 void alina.sim.Simulation.solveStep (double *dt*)

Cette méthode permet de faire avancer la simulation durant un intervalle de temps (très petit). Elle utilise les valeurs de l'état actuel du système pour calculer et stocker les nouvelles valeurs de l'état, après un laps de temps très court donné en paramètre. En fait, cette méthode résout numériquement le système d'équations différentielles ordinaires qui gouvernent le mouvement du pendule inversé.

Paramètres

dt	le laps de temps entre deux calculs successifs de l'état du système (en s)

3.12.4 Documentation des données membres

3.12.4.1 final double alina.sim.Simulation.g = 9.81 [static]

L'accélération gravitationnelle.

3.12.4.2 int alina.sim.Simulation.precision = 1000 [static], [private]

La "précision" du calcul numérique (i.e. le nombre de sous-intervalles temporaires égaux dt pour 1 ms).

3.12.4.3 State alina.sim.Simulation.startState

L'état de départ de la simulation.

3.12.4.4 State alina.sim.Simulation.state

L'état courant de la simulation.

3.12.4.5 Strategy alina.sim.Simulation.strategy

La stratégie utilisée pour la simulation.

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/Simulation.java

3.13 Référence de la classe alina.sim.Simulation.State

Fonctions membres publiques

— State ()
— State (State state)

Attributs publics

— double t
— boolean started
— boolean failed
— double m
— double M
— double I
— double theta
— double vAng
— double aAng
— double x
— double vx
— double ax
— double y
— double vy
— double ay

3.13.1 Description détaillée

Cette classe interne correspond à l'état du système à un instant donné (et permet de stocker des informations comme la position, l'angle, les vitesses, les accélérations, ...). Elle est publique et statique et peut être accédée à l'extérieur de la classe englobante grâce à la syntaxe Simulation.State.

3.13.2 Documentation des constructeurs et destructeur

3.13.2.1 alina.sim.Simulation.State.State ( )

Constructeur sans arguments (qui laisse tous les champs initialisés avec les valeurs par défaut implicites).

3.13.2.2 alina.sim.Simulation.State.State (State *state*)

Constructeur par recopie (qui crée un clône de son argument).

Paramètres

state	l'état qui est clôné

3.13.3 Documentation des données membres

3.13.3.1 double alina.sim.Simulation.State.aAng

L'accélération angulaire (°/s^2).

3.13.3.2 double alina.sim.Simulation.State.ax

L'accélération horizontale du chariot (m/s^2).

3.13.3.3 double alina.sim.Simulation.State.ay

L'accélération verticale du chariot (m/s^2).

3.13.3.4 boolean alina.sim.Simulation.State.failed

Marqueur booléen pour savoir si la simulation a échoué ou pas.

3.13.3.5 double alina.sim.Simulation.State.l

La longueur de la tige du pendule (m).

3.13.3.6 double alina.sim.Simulation.State.m

La masse du pendule inversé (kg).

3.13.3.7 double alina.sim.Simulation.State.M

La masse du chariot (kg).

3.13.3.8 boolean alina.sim.Simulation.State.started

Marqueur booléen pour savoir si la simulation est démarrée ou pas.

3.13.3.9 double alina.sim.Simulation.State.t

Le temps courant (s).

3.13.3.10 double alina.sim.Simulation.State.theta

L'angle formé entre la verticale et la tige du pendule (°).

3.13.3.11 double alina.sim.Simulation.State.vAng

La vitesse angulaire (°/s).

3.13.3.12 double alina.sim.Simulation.State.vx

La vitesse horizontale du chariot (m/s).

3.13.3.13 double alina.sim.Simulation.State.vy

La vitesse verticale du chariot (m/s).

3.13.3.14 double alina.sim.Simulation.State.x

L'abscisse du (centre de masse) du chariot (m).

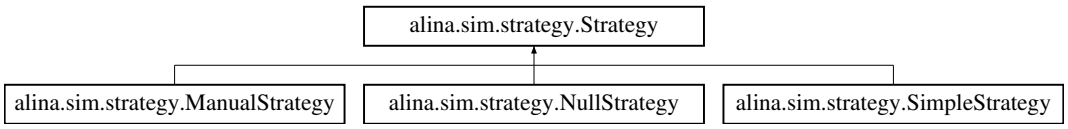
3.13.3.15 double alina.sim.Simulation.State.y

L'ordonnée du (centre de masse) du chariot (m).

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/Simulation.java

3.14 Référence de l'interface alina.sim.strategy.Strategy

Graphe d'héritage de alina.sim.strategy.Strategy :



Fonctions membres publiques

— void reset ()
— double react (Simulation.State state)
— JPanel getPanel ()

3.14.1 Description détaillée

Interface publique qui doit être implémentée dans toutes les classes de définissant des stratégies de stabilisation du pendule inversé et qui doivent (re)définir les méthodes ci-dessous.

Auteur

alina petrescu

Version

1.0

3.14.2 Documentation des fonctions membres

3.14.2.1 JPanel alina.sim.strategy.Strategy.getPanel ( )

Cette méthode retourne le panneau de paramétrage de la stratégie.

Renvoie

le panneau de la simulation

Implémenté dans alina.sim.strategy.SimpleStrategy, alina.sim.strategy.ManualStrategy, et alina.sim.strategy.Null↩ Strategy.

3.14.2.2 double alina.sim.strategy.Strategy.react (Simulation.State *state*)

Cette méthode doit calculer la force à appliquer par le "moteur" du chariot pour tenter de garder le pendule inversé en équilibre.

Paramètres

state	l'état actuel du système physique chariot-tige-pendule

Renvoie

la force à appliquer sur le chariot

Implémenté dans alina.sim.strategy.SimpleStrategy, alina.sim.strategy.ManualStrategy, et alina.sim.strategy.Null↩ Strategy.

3.14.2.3 void alina.sim.strategy.Strategy.reset ( )

Cette méthode permet de réinitialiser la stratégie.

Implémenté dans alina.sim.strategy.SimpleStrategy, alina.sim.strategy.ManualStrategy, et alina.sim.strategy.Null↩ Strategy.

La documentation de cette interface a été générée à partir du fichier suivant :
— src/alina/sim/strategy/Strategy.java

3.15 Référence de la classe alina.sim.ui.StrategyPanel

Est dérivée de JPanel.

Fonctions membres publiques

— StrategyPanel (Strategy[ ] strategies)
— void show (int index)

3.15.1 Description détaillée

Cette classe affiche en bas à droite ce que la stratégie sélectionnée nous permet de faire.

Auteur

alina petrescu

Version

1.0

3.15.2 Documentation des constructeurs et destructeur

3.15.2.1 alina.sim.ui.StrategyPanel.StrategyPanel (Strategy[ ] *strategies*)

Cette méthode (constructeur public) nous affiche les éléments de la stratégie.

Paramètres

strategies	tableau de toutes les stratégies disponibles

3.15.3 Documentation des fonctions membres

3.15.3.1 void alina.sim.ui.StrategyPanel.show (int *index*)

Cette méthode permet de sélectionner la stratégie.

Paramètres

index	l'index de la stratégie voulue

La documentation de cette classe a été générée à partir du fichier suivant :
— src/alina/sim/ui/StrategyPanel.java

Annexe B

Paramètres modifiables

Afin de pouvoir utiliser pleinement le programme élaboré dans le cadre de ce travail de Bachelor, il est important de revoir les principaux paramètres qui peuvent être modifiés soit par l'utilisateur final grâce à l'interface graphique mise à sa disposition, soit par le programmeur grâce à des modifications du code source. En combinant convenablement ces paramètres, on peut créer les cas d'étude qui nous intéressent et, de plus, adapter la simulation numérique et, plus particulièrement, les représentations graphiques associées de manière adéquate pour chaque cas à part.

Plus précisément, l'utilisateur final peut choisir :

- les valeurs des grandeurs géométriques et mécaniques qui caractérisent le système physique (la masse du pendule, la masse du chariot, la longueur de la tige et l'angle initial de déviation par rapport à la verticale, grâce aux curseurs *Pendulum mass*, *Cart mass*, *Length* et, respectivement, *Angle*) ;
- la stratégie de stabilisation du mouvement du pendule inversé (grâce à la liste de choix *Select Strategy*), ainsi que différentes valeurs caractéristiques à la stratégie choisie (grâce à des curseurs appropriés affichés dynamiquement dans la région *Strategy*) ;
- la précision du calcul numérique (grâce au curseur *Simulation precision*) ;
- la vitesse de déroulement de la simulation (grâce au curseur *Speed*).

À son tour, le programmeur peut modifier toute une série de paramètres disponibles, le plus souvent, sous la forme de champs statiques et finaux de différentes classes :

- dans la classe *Simulation* :
  - l'accélération gravitationnelle (`g = 9.81` m/s^2) ;
- dans la classe *MainPanel* :

  - le nombre d'images à afficher par seconde (`FRAMES_PER_SECOND = 50`) ;
- dans la classe *AnimationPanel* :
  - l'échelle des longueurs (`SCALE = 100` pixels/m) ;
  - la largeur initiale du chariot (`CART_WIDTH = 0.4` m) ;
  - la hauteur initiale du chariot (`CART_HEIGHT = 0.2` m).

De plus, toujours par l'intermédiaire de l'interface graphique, l'utilisateur final peut :

- démarrer, arrêter et redémarrer une simulation en cours (grâce au bouton `PLAY/PAUSE`) ;
- exécuter une simulation pas à pas (grâce au bouton `Step`) ;
- réinitialiser une simulation (grâce au bouton `Reset`).

Bibliographie

[1] Union des Professeurs de Physique et de Chimie. *Stabilisation d'un équilibre instable*. http://slideplayer.fr/slide/3250499/, 2014. France.

[2] Freddy Mudry. *Modélisation et régulation d'un pendule inversé*. http://freddy.mudry.org/public/NotesApplications/na_pendinvc.pdf, 2003. École d'Ingénieurs du canton de Vaud.

[3] Tarek Kerir et Geoffroy Kirstetter et Jeanne Molinari. *La dynamique du jongleur ou comment construire une otarie mécanique*. http://physique.unice.fr/sem6/2007-2008/PagesWeb/Irh/l_otarie_mecanique/accueil.html, 2007-2008. Université de Nice Sophia Antipolis.

[4] Wikipedia. *Kapitza's pendulum*. https://en.wikipedia.org/wiki/Kapitza's_pendulum, 2016.

[5] Dimitri van Heesch. *Doxygen*. http://www.stack.nl/~dimitri/doxygen/. 1.8.12.

[6] Alfio Maria Quarteroni et Fausto Saleri et Paola Gervasio. *Calcul Scientifique - cours, exercices corrigés et illustrations en Matlab et Octave*. 2010. Springer, Milano.

[7] Nicolas Petit. *Automatique - Dynamique et Contrôle des systèmes*. https://sgs.mines-paristech.fr/prod/sgs/ensmp/catalog/course/detail.php?code=c1422&lang=FR&type=DETAIL&year=1A, 2015. École d'ingénieurs MINES ParisTech.

[8] Jean-Philippe Roberge. *Introduction à l'automatisation*. http://slideplayer.fr/slide/448874/, 2011. École Polytechnique de Montréal.

[9] S. Mottelet. *Automatique Avancée - Introduction à la commande des systèmes non-linéaires*. 2013. Université de Technologie de Compiègne.

[10] Frédéric Bonnans et Pierre Rouchon. *Analyse et commande de systèmes dynamiques*. `http://cas.ensmp.fr/~rouchon/x03/poly03.pdf`, 2003. École d'ingénieurs MINES ParisTech.

[11] Tiobe. `http://www.tiobe.com/tiobe-index/`.

[12] Claude Delannoy. *Programmer en Java*. Eyrolles, 2014.

[13] Object Technology International. *Eclipse*. `https://eclipse.org/`, 2016. version 4.6.1 - neon.1.