# 2 Tracé de la courbe théorique

## Objectifs

Il s'agit de développer le programme de calcul et de visualisation de la trajectoire théorique de la masse dans les phases I (moteur en marche) et II (moteur interrompu), étant donnés :
- la formulation mathématique de la trajectoire $x(t)$ (l'ordonnée de la masse en fonction du temps) ;
- les valeurs des paramètres physiques du dispositif (masse, raideur du ressort,...) entrant dans cette formulation et que vous aurez mesurées sur votre propre montage ;
- la classe `Matrice` pour le calcul de $x(t)$ ;
- un programme Java d'affichage de courbes entièrement fourni

## Vérification de l'installation

Le programme Java pour l'affichage des courbes comprend quatre fichiers :

- `Courbe.java` permet de représenter une courbe par un tableau de points *(abscisse, ordonnée)* ;
- `Graphique.java` permet d'afficher sur un graphique des objets de type `Courbe` ;
- `marques.png` (utilisé par `Graphique` $\longrightarrow$ *à placer dans le répertoire d'où le programme est lancé*)
- `FichierPNG.java` (utilisé par `Graphique` $\longrightarrow$ *à placer dans le répertoire d'où le programme est lancé*)

**Avant de commencer** :

Compilez les fichiers `Courbe.java` et `Graphique.java` puis exécutez `Graphique` pour tester votre installation.

## 2.1 Initialisation des matrices et vecteurs

En mathématiques, vous avez déterminé la solution $Y(t) \in \mathbb{R}^3$ du système d'équations différentielles pour les phases 1 (moteur en marche) et 2 (moteur stoppé). Dans les deux cas, la solution peut s'exprimer comme suit :

$$Y(t) = e^{tM}.(Y(t_{init}) - Y_{SP}(t_{init})) + tY'_{SP} + Y_{SP}(t_{init}) \quad \text{avec} \quad Y(t) = \begin{pmatrix} x(t) \\ \dot{x}(t) \\ X(t) \end{pmatrix} \tag{2}$$

où les matrices et vecteurs

$$M, \quad Y_{SP}(t_{init}) \quad \text{et} \quad Y'_{Sp}$$

dépendent du vecteur de conditions initiales $Y(t_{init})$ et des paramètres suivants du dispositif :

$$\omega_0^2, \quad \beta, \quad V, \quad X(0).$$

Notez que $X(0)$ est la position du point d'attache au démarrage (à ne pas confondre avec $X(t_{init})$).

La classe `Solution` a pour rôle de calculer $Y(t)$ en fonction de $t$ une fois les paramètres ci-dessus connus. Ces paramètres lui sont donc fournis dans son constructeur.

**Question 1**.

Implémentez le constructeur de la classe `Solution` qui doit initialiser les matrices `M`, `YSP_prime` ($Y'_{SP}$) et `YSP_init` ($Y_{SP}(t_{init})$) en fonction des arguments : `Y_init` ($Y(t_{init})$), $\omega_0^2$ $\beta$, $V$ et $X(0)$. Lisez l'astuce ci-dessous.

La syntaxe suivante pourra simplifier votre travail :

```
double[][] tab = {{0,1},{1,0},{0,0}};          // déclaration+allocation+initialisation
```

Puis :

```
Matrice M = new Matrice(tab);                  // création de la matrice
```

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}$$

Attention, on ne peut pas découper la première ligne en deux étapes :

```
double[][] tab; tab = {{0,1},{1,0},{0,0}};     // NON : erreur de syntaxe
```

## 2.2 Calcul de la solution

Nous pouvons maintenant écrire la méthode qui retourne l'état du système, le vecteur $Y$, à un instant $t$ (quelle que soit la phase courante).

**Question 2**.

Implémentez dans la classe `Solution` les méthodes suivantes :
- `Y(double t)` qui retourne le vecteur $Y(t)$ d'après la formule (2)
- `x(double t)` qui retourne le premier élément du vecteur $Y(t)$

## 2.3 Calcul des paramètres

Les 4 paramètres $\omega_0^2$ $\beta$, $V$ et $X(0)$ peuvent en fait être calculées à partir de paramètres plus élémentaires, certains étant supposés connus exactement (comme la gravitation), d'autres soumis à des incertitudes liées aux mesures (comme la masse). Ils dépendent également de la phase de la trajectoire dans laquelle on se trouve.

La classe `CourbeTheo` a pour rôle de construire entièrement la courbe. Elle calcule pour cela la phase courante, détermine la valeur des paramètres ci-dessus et invoque `Solution` pour finalement obtenir les points.

On considère donc dans la classe `CourbeTheo` trois types de paramètres :

1. les paramètres d'entrée : ce sont les paramètres physiques en « entrée » du modèle comme, par exemple, la masse. Ces paramètres ne peuvent être calculés à partir d'autres paramètres et peuvent être amenés à être modifiés (la masse n'étant connue qu'approximativement).
2. les paramètres fixés : ce sont les paramètres que l'on connaît et qui ne seront jamais modifiés.
3. les paramètres calculés : ce sont les paramètres qui serve au calcul de la solution (et que l'on calcule à partir des précédents). **Ils correspondent aux variables d'instance de la classe** `CourbeTheo`.

**Question 3**.

Implémentez le constructeur de la classe `CourbeTheo` qui initialise toutes les variables d'instance. Utilisez pour cela les formules mathématiques reliant les différents paramètres.

## 2.4 Tracé de la trajectoire

Pour tracer la trajectoire complète, on calculera la position $x(t)$ de la masse à différents instants $t_0, \ldots, t_n$. C'est la méthode `ajouterPoint(double t)` qui va se charge pour des temps $t$ entrés successivement d'ajouter dans la courbe les points $(t, x(t))$.

C'est cette méthode qui va **gérer le changement de phase**. Cela signifie qu'on doit obtenir la courbe simulant la dynamique du monte-charge en appelant simplement de façon répétée cette méthode pour des temps $t_0, \ldots, t_n$.

En contre-partie, la méthode impose que les temps soient entrés de façon croissante de telle sorte qu'elle puisse « basculer » d'une phase à l'autre, en guettant le moment où la barrière lumineuse est franchie. Utilisez la variable d'instance `phase1` pour mémoriser la phase en cours.

**Question 4**.

Implémentez dans la classe `CourbeTheo` la méthode `ajouterPoint(double t)` qui ajoute dans la courbe le point d'abssice $t$ et d'ordonnée $x(t)$ (en provoquant éventuellement un changement de phase).

Les appels à cette méthode ne pourront être faits que pour des $t$ croissants.

Attention : souvenez-vous que le changement de phase repose aussi sur le vecteur `Y_init`.

Il ne nous reste plus qu'à générer une séquence d'appels à `ajouterPoint` en discrétisant un intervalle de temps.

**Question 5**.

- Implémentez dans la class `CourbeTheo` la méthode `ajouterPoints(double tmin, double tmax, double pas)` qui crée la courbe théorique avec des points calculés pour $t$ variant entre `tmin` et `tmax`, avec le pas `pas`.
- Créez une courbe dans le `main` de la classe `Main` sur l'intervalle de temps $[0, 5]$ avec un pas de $0.01s$.
- Affichez la courbe et observez le changement de phase.

# 3 Validation

## Objectifs

Le but ici est de valider notre travail fait sur la classe `CourbeTheo`, en confrontant (graphiquement) la courbe obtenue avec

1. une courbe obtenue par une formule « explicite » (sans exponentielle de matrices)
2. la courbe expérimentale

## 3.1 Confrontation avec la courbe analytique

Nous allons créer une nouvelle classe qui aura pour simple but de construire la courbe en utilisant l'une des formes analytiques que vous aurez obtenues pendant les séances de maths. Cette classe est beaucoup plus simple que `CourbeTheo` car elle ne nécessite pas de manipuler des matrices ni de définir de nouvelles fonctions mathématiques comme l'exponentielle de matrice. Toutes les opérations usuelles (sin,tan,cosh, etc.) existent en effet déjà dans la classe native `Math`.

La forme analytique peut correspondre à un cas particulier, comme celui où $\beta = 0$.

Il faut s'assurer, en revanche, que ce cas particulier permette bien d'observer (donc de valider) le calcul d'exponentielle effectué dans `CourbeTheo` (c'est le cas avec $\beta = 0$ !).

**Question 1**.

1. Créez la classe `CourbeAnalytique` qui étend `Courbe`
2. Implémentez la méthode `ajouterPoints(double tmin, double tmax, double pas)` répondant aux mêmes spécifications qu'à la question 2-(5).
3. Modifiez le `main` de la classe `Main` pour faire afficher les deux courbes.

### 3.2 Confrontation avec la courbe expérimentale

Nous allons maintenant charger la courbe expérimentale, la synchroniser avec la courbe théorique et constater visuellement leur faible adéquation.

Une classe `CourbeExpe.java` vous est fournie. Elle permet de construire une courbe à partir de mesures contenues dans un fichier. Pour savoir comment l'utiliser, consultez sa documentation (javadoc). Vous verrez notamment qu'il est possible de modifier l'origine des temps, ce qui vous permettra de rectifier le décalage entre le chronomètre de la vidéo et le temps $t = 0$ de votre expérience.

**Question 2**.

1. Construire Dans le `main` de la la classe `Main.java` une courbe à partir de votre fichier de mesure
2. Implémentez dans la class `CourbeTheo` la méthode `ajouterPoints(Courbe cref)` qui crée la courbe théorique avec des points calculés avec les abssices d'une autre courbe.
3. Régler (visuellement) l'origine des temps de la courbe expérimentale pour qu'elle soit synchronisée avec la courbe théorique

Si le calcul de votre courbe théorique est juste, les deux courbes doivent avoir une « allure » ou un « comportement » similaire. En revanche, vous vous appercevrez sans doute que, dans le détail, elles sont bien différentes.

Les écarts observés sont tels qu'ils ne peuvent être imputés au modèle mathématique (le fait d'avoir négligé les forces de frottement, par exemple) ou aux imprécisions des mesures. Ils sont dû à des paramètres physiques du système (masse, raideur, etc.) que l'on a mal identifiés.

L'outil informatique va nous permettre maintenant de proposer automatiquement de nouvelles valeurs pour ces paramètres de telle sorte que les courbes théoriques et expérimentales se juxtaposent le mieux possible.

Votre but dans la section suivante est de compléter ce mécanisme d'ajustement.

## 4 Ajustement des paramètres

### Objectifs

1. implémenter un *critère d'inadéquation* d'une courbe par rapport à une courbe de référence, critère basé sur un calcul de distance.
2. implémenter un algorithme ajustant automatiquement les paramètres physiques de telle sorte que la courbe théorique obtenue minimise l'inadéquation.
3. observer l'amélioration obtenue

## 4.1 Initialisation

Un squelette de classe `Ajusteur` vous est fourni. Son principe est le suivant.

On considère tout d'abord qu'il y a `CourbeTheo.NB_PARAMS` paramètres (ici, 7) qui sont susceptibles d'être ajustés. Ils correspondent aux paramètres d'« entrée » de la classe `CourbeTheo`.

Il est clair qu'il est pratique de manipuler ce jeu de paramètres dans un unique tableau, ce qui explique pourquoi la classe `CourbeTheo` possède un constructeur prenenant en argument un tableau de paramètres.

Un ajusteur est construit avec une valeur initiale du tableau de paramètres.

Comme l'ajusteur va chercher à modifier le jeu de paramètres jusqu'à obtenir une bonne adéquation avec la courbe expérimentale, cette dernière sert donc de courbe de **référence**. Elle est donc également transmise au constructeur.

**Question 2**.

Ajoutez dans le `main` de `Main.java` une ligne qui crée un ajusteur avec comme vecteur de paramètres initial : vos mesures.

## 4.2 Critère d'inadéquation

Lorsque l'ajusteur essaye un nouveau jeu de paramètres, il doit mesure l'adéquation entre la courbe théorique qu'il calcule avec ce jeu et la courbe expérimentale. Il nous faut donc un moyen de calculer la *distance* entre deux courbes $(t, f_1(t))$ et $(t, f_2(t))$ définie sur un même intervalle $[t]$. L'aire qui les sépare est le candidat le plus naturel :

$$distance(f_1, f_2) = \int_{[t]} |f_1(t) - f_2(t)| dt \tag{3}$$

**Question 3**.

Implémentez dans la classe `Courbe` la méthode `distance(Courbe c2)`.

- On suppose que les courbes ont les même abscisses
- La méthode doit retourner une valeur approchée de la distance définie par (3).

## 4.3 Un pas en avant, un pas en arrière...

Maintenant que notre critère à minimiser est défini (la distance), l'ajusteur possède un moyen d'évaluer la qualité d'un jeu de paramètres. Le principe de l'ajusteur est alors simple : modifier (légèrement) le jeu de paramètres pour faire décroître le critère. Bien sûr, il ne peut pas savoir à l'avance comment modifier les paramètres : il « tâtonne ». Plus précisément, il ne modifie que la valeur d'un seul paramètre à la fois. Il commence par regarder s'il vaut mieux diminuer le paramètre (aller à gauche) ou l'augmenter (aller à droite) pour améliorer le critère. S'il faut se déplacer à gauche, il se déplace tant que cela permet d'améliorer le critère. De même, s'il faut se déplacer à droite, il le fait autant que possible.

C'est ce que doit effectuer la méthode `deplacer`.

**Question 4**.

Implémentez la méthode `deplacer(int i, boolean sens)` de la classe `Ajusteur`.
- l'argument `i` est le numéro du paramètre à modifier
- l'argument `sens` vaut `GAUCHE` ou `DROITE`.

La méthode s'appuie sur les variables d'instance `p`, `dist`,`inf`, `pas` et `sup` :
- `p` contient **le meilleur jeu de paramètres** obtenu par l'ajusteur depuis sa création. Si la méthode parvient à améliorer le critère, elle doit modifier `p` en conséquence.
- `dist` contient la valeur du critère pour `p`. Si `p` est modifié, `dist` doit est maintenu à jour.
- le $i^{eme}$ paramètre `p[i]` ne peut être déplacé que de `pas[i]` et doit toujours vérifier

$$\texttt{inf[i]} \leq p_i \leq \texttt{sup[i]}$$

## 4.4 Le point fixe

Il reste à implémenter la méthode principale, `ajuster`. Son principe consiste à déplacer le premier paramètre, puis le second et ainsi de suite jusqu'au dernier. Cependant, à l'issue de ce simple balayage, il n'est pas encore garanti qu'un minmum soit atteint. En effet, comme cela est illustré sur la figure suivante, le fait de déplacer un paramètre $p_2$ peut "libérer" un paramètre $p_1$ que l'on avait déjà traité, c.a.d. que $p_1$ peut de nouveau permettre une diminution du critère.

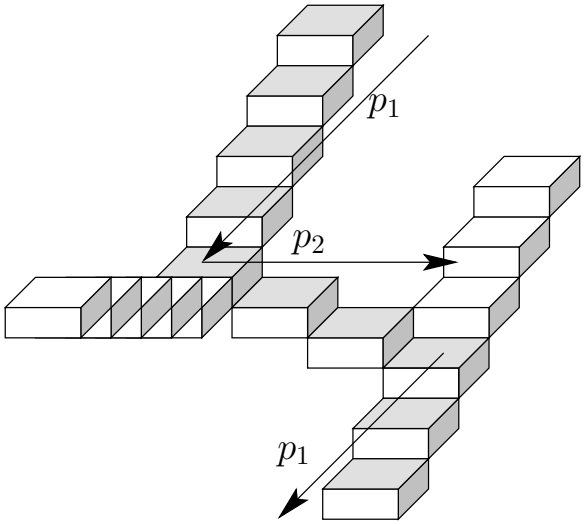


FIGURE 1 – Principe de la minimisation en « escalier ». On descend tout d'abord suivant le paramètre $p_1$. Une fois le palier atteint, on descend suivant le paramètre $p_2$. Il alors de nouveau possible de descendre suivant $p_1$.

**Question 5**.

Implémentez la méthode `ajuster` de la classe `Ajusteur` en **ignorant** pour le moment l'argument `start`. Votre algorithme doit commencer la minimisation par le paramètre `p[0]` et boucler jusqu'à ce que plus aucun paramètre ne puisse diminuer le critère.

## 4.5 Un peu de globalité

L'algorithme que l'on a écrit converge bien vers un minimum [1], mais ce minimum est *local* : c'est un « creux » parmi d'autres (imaginer une boîte d'oeufs), sachant que les creux peuvent être plus ou moins profonds.

1. La convergence nécessite en fait que le critère soit différentiable (mais c'est le cas ici).

On comprend intuitivement que le premier paramètre sur lequel on décide de se déplacer est très déterminant : c'est en gros ce paramètre qui va décider du creux vers lequel on se dirige.

Afin de captuer les différents creux proches de nos valeurs initiales, l'idée est de modifier `ajuster` pour qu'elle commence sa recherche par un paramètre `p[start]` (au lieu de `p[0]`).

**Question 6**.

1. Modifier `ajuster` pour qu'elle prenne en compte le paramètre `start`.
2. Reprenez le `main` (de la classe `Main`), là où vous aviez créé un ajusteur. Faites à la place une boucle qui crée un ajusteur et calcule un point fixe en commençant à chaque fois par un paramètre différent. Faites afficher à chaque fois la distance finale obtenue et le jeu de nouveaux paramètres candidat. Observez.
3. Comparez la courbe expérimentale avec la courbe théorique calculée avec votre meilleur candidat.