Projet intégrateur : informatique

2nde thématique transversale

Comportement d'un lève-charge

CB1 2007-2008

1 Objectifs et déroulement de la session informatique

1.1 Objectifs

Pour la seconde thématique transversale du projet intégrateur, l'objectif de la session informatique vise à réaliser un **outil de calcul et de simulation** de la trajectoire d'une masse suspendue à un système de lève-charge. Grâce à cet outil, vous déterminerez le premier instant auquel la trajectoire de la masse coupe théoriquement la barrière lumineuse de votre montage. Vous pourrez alors reconstituer la trajectoire complète de la masse : pendant son élévation puis après l'interruption du lève-charge.

La session consiste en la **réalisation individuelle d'une application informatique**, depuis la conception des algorithmes jusqu'à leur implémentation dans le langage Java.

1.2 Organisation de la session et rendu des travaux

Vous disposez de plus de 3 semaines pour la réalisation de ce projet, jalonnées comme suit :

pour le 14 mars lire l'énoncé du TP1 (section 2) ; organisez votre répertoire de travail (**E1**, section 2.2) : seul le fichier `PreTrajectoire.java` ne sera disponible sur Campus qu'au début du TP1.

14 mars première séance de **TP** – apportez votre polycopié **PROGRAMMATION** : conception, implémentation et test des algorithmes ; code source à compléter `PreTrajectoire.java` ;

pour le 21 mars **travail personnel requis** : terminer `PreTrajectoire.java` ;
faire une copie nommée `Trajectoire.java` ; dans ce fichier, remplacer toutes les occurences de `PreTrajectoire` par `Trajectoire` ; nettoyer la classe `Trajectoire` : supprimer les méthodes nommées `factorielle`, `sommeFactorielleNaif`, `sommeFactorielle`, `sommeExposant`, `ajouteSerie`, et leurs appels dans la méthode `main` ;
réfléchir à l'adaptation de la classe `Trajectoire` dans le cadre de la problématique du lève-charge ; rassembler les paramètres nécessaires aux calculs.

21 mars seconde séance de **TP NOTÉ** – apportez votre polycopié **PROGRAMMATION** : procéder à cette adaptation ; code source à modifier/compléter `Trajectoire.java` ;

21-31 mars fin de l'implémentation (+ améliorations facultatives proposées), validation et nettoyage du code ;

31 mars – 20h date **au plus tard** de dépôt de l'archive complète du projet sur Campus.

Vous disposez jusqu'au 28 mars, sur Campus, d'un espace pour poser vos questions : Forum Questions Informatique. Vous pouvez vous y abonner pour recevoir, par mail, les messages de ce forum :

https://nte.gemtech.fr/campus/mod/forum/view.php?id=10102

Les fichiers de code source devront être déposés sur Campus, dans l'espace intégrateur 1ère année. Le retour des évaluateurs se fera aussi par l'intermédiaire de Campus. Prière de tenir compte de ces commentaires...

1.3 Évaluation

La note de la partie informatique du projet intégrateur (2nde thématique transversale) comprend :
- l'évaluation des fichiers `PreTrajectoire.java` et `Trajectoire.java`, remis au TP2 (30%)
- l'évaluation de l'application finale (70%)

Ces évaluations reposeront principalement sur les critères suivants :
- **respect des conditions de remise des devoirs** : aucun devoir ne sera accepté au-delà des dates et heures limites. Prenez vos précautions en cas de surcharge de Campus : n'attendez pas le dernier moment et pensez à déposer une version préliminaire de votre projet quelque temps avant l'heure limite.
- **conformité du programme** : tout programme rendu doit, *a minima*, compiler et s'exécuter. Les parties de code développées en TP, non terminées ou qui ne compileraient pas à l'issue de la séance, devront être placées en commentaire. L'application finale doit être validée par les tests.
- **justesse de la programmation et des algorithmes** : appel de méthodes, emploi des variables, passage de paramètres, parcours de tableaux, emploi des boucles, etc.
- **qualité (clarté) du code** : les règles de programmation énoncées dans les cours de programmation et d'algorithmique doivent être respectées.

Rappel : **la fraude et la copie seront lourdement sanctionnées** (=0, sans préjuger d'éventuelles suites disciplinaires).

1.4 Groupes et enseignants

Répartition des élèves suivant les groupes APA : Philippe David (A), Sophie Demassey (B), Rémi Douence (C), Jérôme Fortin (D), Assia Hachichi (E), Nicolas Loriant (F).
Resp : Sophie.Demassey@emn.fr, bureau B210.

2 TP1 : conception et implémentation des algorithmes

2.1 Objectif et déroulement de la séance

Vendredi 14 mars 2008, 2h.

Avant de débuter le développement de l'application à proprement dite, vous concevrez et implémenterez, lors de cette séance, les différents algorithmes qui seront mis en œuvre et les testerez sur des exemples de valeurs. Ces algorithmes consistent principalement en des calculs de sommes partielles et d'expressions polynomiales sur des matrices. Le squelette de la classe `PreTrajectoire`, dans laquelle vous implémenterez ces algorithmes, est fournie, ainsi qu'une classe `Matrice`, complète elle, qui fournit l'ensemble des opérations requises pour la manipulation des matrices (création, somme, produit).

Comme les calculs à effectuer résultent parfois en de très grands nombres (supérieurs à 10^{300}), cet exercice vous donnera l'occasion d'observer un principe fondamental en informatique : *l'infini n'existe pas*. En effet, comme l'espace mémoire d'un ordinateur est limitée, il n'est pas possible de manipuler des données numériques arbitrairement grandes. Cependant, vous verrez que choisir un type de données adéquate ou implémenter un algorithme *intelligent* permet de repousser cette limite.

Si vous souhaitez recevoir un commentaire sur votre code, vous avez la possibilité de déposer sur Campus votre fichier `PreTrajectoire.java` à l'issue de la séance (avant 20h ce jour même). **E0**

2.2 Organisez votre répertoire de travail

Comme pour le projet précédent, votre application comprend plusieurs fichiers (code source, bibliothèques, fichier de configuration) ; aussi il est préférable d'organiser le répertoire la contenant :

1. créez le répertoire du projet `th2_apa_X` (remplacez `X` par votre nom), et ses trois sous-répertoires : `classes` (pour les fichiers compilés `.class`), `lib` (pour les archives Java `.jar`) et `src` (pour les fichiers source `.java`)
2. téléchargez depuis Campus (ou recopiez-les du projet 1) :
   - les **fichiers source** `PreTrajectoire.java` et `TraceSeries.java`
   - le **fichier compilé** `Matrice.class`
   - les **archives Java** `jfreechart.jar` et `jcommon.jar`

   et enregistrez-les dans les sous-répertoires correspondants.
3. compilez les fichiers sources et exécutez `PreTrajectoire`.

E1

Rappel des commandes sous Windows :

```
javac -d classes -cp classes;lib\jfreechart.jar;lib\jcommon.jar src\*.java   compile
java -cp classes;lib\jfreechart.jar;lib\jcommon.jar PreTrajectoire           exécute
```

Rappel des commandes sous Linux/MacOS :

```
mkdir rep                                                                    crée le répertoire rep
javac -d classes -cp classes:lib/jfreechart.jar:lib/jcommon.jar src/*.java   compile
java -cp classes:lib/jfreechart.jar:lib/jcommon.jar PreTrajectoire           exécute
```

Rappel sur la signification des commandes :

- l'option `-d` *rep* stipule que les fichiers compilés seront placés dans le répertoire *rep*
- l'option `-cp` *rep1;rep2;...* indique les chemins d'accès aux bibliothèques (*classpath*)
- les noms des répertoires et archives Java contenant ces bibliothèques sont séparés par (`;`) sous Windows ou (`:`) sous Linux
- `*.java` est une expression régulière signifiant « tous les fichiers dont le nom se termine par `.java` »

2.3 La classe `PreTrajectoire`

C'est dans cette classe, dont le squelette vous est fourni, que vous allez implémenter et tester les onze algorithmes. À chaque algorithme correspond une unique méthode de la classe :

Partie 1 : séries, complexité, et problèmes numériques

```
int factorielle(int)
int sommeFactorielleNaif()
long sommeFactorielle()
double sommeExposant(double)
Matrice sommeExposant()
Matrice sommeExponentielle(double)
```

Partie 2 : calcul des fonctions matricielles

```
Matrice combinaisonMatrices(Matrice, Matrice, Matrice, Matrice, double)
Matrice Y(double)
```

Partie 3 : tracé des courbes

```
void ajouteSerie(TraceSeries)
void ajouteSerieTronquee(TraceSeries)
void traceCourbe()
```

Pour chaque méthode, vous trouverez dans le fichier `PreTrajectoire.java` : les spécifications de la méthode (ce qu'elle fait et comment elle s'utilise), la signature de la méthode et le bloc avec, éventuellement, une instruction de retour **à remplacer**[1], par exemple :

```
/**
 * calcul de la factorielle de j        fonction
 * @param j entier positif              noms et types des paramètres en entrée
 * @return 1x2x..xj entier              valeur et type de la donnée de retour
 */
public int factorielle(int j) {
      return 1;                         instruction à modifier
}
```

La procédure de test de vos algorithmes est détaillée dans la méthode `main`. Il s'agit de vérifier (par simple affichage) que chaque méthode retourne bien la valeur attendue sur des exemples de valeurs d'entrées donnés. Des questions concernant la complexité des algorithmes sont aussi posées. Vous pouvez y répondre par un commentaire dans le code. Par exemple :

```
public static void main (String[] args) {
      PreTrajectoire T = new PreTrajectoire();

      // Q1 : implementer factorielle(int)
      //      verifier : T.factorielle(0) == 1
      System.out.println("facto(0) : " + T.factorielle(0));

      //      combien d'operations sont effectuees par la methode factorielle(10) ?
      // REPONSE : 9
}
```

[1] les instructions de retour sont nécessaires à rendre le fichier compilable en l'état

Compléter `PreTrajectoire.java`, en suivant les instructions détaillées dans la méthode `main` :

Q1 : implementer : factorielle(int)
verifier : T.factorielle(0) == 1
verifier : T.factorielle(10) == 3628800
combien d'operations (multiplication d'entiers) sont effectuees par la methode factorielle(10) ?
combien d'operations (multiplication d'entiers) sont effectuees par la methode factorielle(j), j>1 ?
Q2 : implementer : sommeFactorielleNaif()
(T.I=10) verifier : T.sommeFactorielleNaif() == 4037914
combien d'operations (somme et multiplication d'entiers) sont effectuees par sommeFactorielleNaif() ?
Q3 : implementer : sommeFactorielle()
(T.I=10) verifier : T.sommeFactorielle() == 4037914
combien d'operations (somme et multiplication d'entiers) sont effectuees par sommeFactorielle() ?
poser I=100 dans le constructeur PreTrajectoire()
verifier : T.sommeFactorielle() == 1005876315485501978
expliquer pourquoi T.sommeFactorielleNaif() ne retourne pas ce resultat ?
facultatif : effectuer les modifications necessaires
Q4 : implementer : sommeExposant(double)
(T.I=100) verifier : T.sommeExposant(0) == 1
verifier : T.sommeExposant(1) == 101
verifier : T.sommeExposant(2) == 2.53530120045645 88E30 > 2.10^{30}
verifier : T.sommeExposant(10) == 1.1111111111111118E100 > 1.10^{100}, expliquer ce chiffre 8 ?
calculer : T.sommeExposant(10000), interpreter
poser I=500 dans le constructeur PreTrajectoire()
noter que sommeExposant(10) n'est plus calculable ainsi
Q5 : implementer : sommeExposant()
(T.I=500, T.M={{1,0,0},{0,1,0},{0,0,1}}, la matrice identite de taille 3 : Id3)
verifier : T.sommeExposant() == 501.Id3
poser M={{0,1,0},{-50,0,50},{0,0,0}} dans initialiseMatrices()
calculer T.sommeExposant(), interpreter
poser M={{0,1,0},{0,0,0},{0,0,0}} dans initialiseMatrices()
verifier : T.sommeExposant() == Id3+M == {{1,1,0},{0,1,0},{0,0,1}}
Q6 : implementer : sommeExponentielle(double)
(T.I=500, T.M={{0,1,0},{0,0,0},{0,0,0}})
verifier : T.sommeExponentielle(0) == Id3
verifier : T.sommeExponentielle(2) == Id3 + 2.M
poser M=Id3 dans initialiseMatrices()
verifier : T.sommeExponentielle(0) == Id3
verifier : T.sommeExponentielle(2) == (7.389056098930649).Id3
Q7 : implementer : combinaisonMatrices(Matrice,Matrice,Matrice,Matrice,double)
declarer et initialiser une matrice 3x3 : B = Id3
declarer et initialiser une matrice 3x1 : C = {{1},{1},{1}}
verifier : T.combinaisonMatrices(B,C,C,C,2) == 4.C
verifier : T.combinaisonMatrices(B,C,C,C,0) == 2.C
Q8 : implementer : Y(double)
(T.I=500, T.M=Id3) verifier : T.Y(0) == combinaisonMatrices(B,C,C,C,0) == 2.C
verifier : T.Y(2) == (10.389056098930649).C
Q9 : implementer : ajouteSerie(TraceSeries) et traceCourbe()
afficher la courbe x(t)
Q10 : implementer : ajouteSerieTronquee(TraceSeries)
modifier traceCourbe() pour afficher la courbe tronquee
faire afficher tBarriere par ajouteSerieTronquee(TraceSeries)
modifier x(double) de sorte que x(t) == le premier element du vecteur Y(t) pour tout t
poser I=100, XBarriere=4.5, M={{0,1,0},{-50,0,50},{0,0,0}}

E2

2.4 Spécifications

2.4.1 `PreTrajectoire`

Certains paramètres d'entrée des méthodes sont définis, non par des arguments de la méthode mais par des variables d'instance, ou encore par une méthode d'instance `x(double)` :

`int I`	rang des sommes partielles (valeur par defaut = 10)
`double tPas`	pas d'abscisse (defaut = 0.01)
`double tMin`	abscisse minimale (defaut = 0)
`double tMax`	abscisse maximale (defaut = 5)
`double xBarriere`	ordonnée maximale (defaut = 4.5)
`Matrice M`	matrice 3×3 (defaut = Identite(3))
`Matrice M1,M2,M3`	vecteurs 3×1 (defaut = {{1},{1},{1}})
`double x(double)`	modélise une fonction croissante $\mathbb{R}^+ \to \mathbb{R}$ (defaut = Id: $t \mapsto t$)

2.4.2 `Matrice`

Cette classe permet de modéliser des matrices de flottants $b = (b_{ij})_{0 \leq i < m, 0 \leq j < n} \in \mathbb{R}^{m \times n}$:

`Matrice(int m, int n)`	constructeur : crée la matrice zéro de $\mathbb{R}^{\mathtt{m} \times \mathtt{n}}$
`Matrice(Matrice a)`	constructeur copie : crée une matrice identique à `a`
`Matrice(double[][])`	constructeur : crée une matrice à partir d'un tableau 2D
`int nbLignes()`	retourne la première dimension de la matrice courante
`int nbColonnes()`	retourne la seconde dimension de la matrice courante
`double getElement(int i, int j)`	retourne l'élément $b_{\mathtt{ij}}$ de la matrice courante
`void setElement(int i, int j, double val)`	affecte `val` à l'élément $b_{\mathtt{ij}}$ de la matrice courante
`static Matrice identite(int n)`	crée et retourne la matrice identité de $\mathbb{R}^{\mathtt{n} \times \mathtt{n}}$
`void ajoute(Matrice a)`	ajoute la matrice `a` à la matrice courante
`void multiplie(double l)`	multiplie la matrice courante par un scalaire `l`
`void multiplieDroite(Matrice a)`	multiplie à droite la matrice courante par `a`
`void multiplieGauche(Matrice a)`	multiplie à gauche la matrice courante par `a`
`static Matrice somme(Matrice a, Matrice b)`	crée et retourne la matrice somme `a+b`
`static Matrice produit(double l, Matrice a)`	crée et retourne la matrice produit `l.a`
`static Matrice produit(Matrice a, Matrice b)`	crée et retourne la matrice produit `ab`
`String toString()`	retourne une chaîne représentant la matrice courante

Rappel Java :

`Matrice a =` **`Matrice.`**`somme(b,c) ;`	appel d'une **méthode de classe** (`static`)
`a.``ajoute(b) ; int m =` **`a.`**`nbLignes() ;`	appel de **méthodes d'instance** sur **a**
`double[][] t = {{0,1},{1,0},{0,0}} ;`	OK : déclaration+allocation+initialisation
`double[][] t ; t = {{0,1},{1,0},{0,0}} ;`	NON : erreur de syntaxe

2.4.3 `TraceSeries`

`TraceSeries()`	constructeur
`void ajouteSerieAna(double[][] tab, String nom)`	ajoute la série « `nom` » de points 2D $(\mathtt{tab[k][0]}, \mathtt{tab[k][1]})_{0 \leq k < \mathtt{tab.length}}$
`void creeGraphique(String nom, String lX, String lY)`	crée un graphique « `nom` » dont les axes sont labellés `lX` (abscisse) et `lY` (ordonnée)
`void afficheGraphique(String nom)`	affiche le graphique dans une fenêtre
`void enregistreGraphique(String nom)`	enregistre le graphique dans `nom.png`

3 TP2 : application à la problématique du lève-charge

3.1 Objectif et déroulement de la séance

Vendredi 21 mars 2008, 2h.

Vous disposez maintenant d'un outil capable de traiter le comportement de votre système de lève-charge :

- déterminer le premier instant auquel la trajectoire de la masse coupe théoriquement la barrière lumineuse ;
- afficher la trajectoire de la masse jusqu'à cet instant.

Il suffit pour cela, d'initialiser les valeurs du programme avec les bonnes valeurs d'entrée.

De manière facultative, vous pourrez également faire afficher la trajectoire de la masse après l'interruption du lève-charge.

Au début de la séance (avant 15h50) : déposez sur Campus le fichier completé `PreTrajectoire.java`. À l'issue de la séance (avant 17h50) : déposez sur Campus le fichier complété `Trajectoire.java`, OU BIEN l'archive complète de votre programme `th2_apa_X.zip` si vos paramètres sont enregistrés dans un fichier à part (`setup.prop`).	**F0**

3.2 Pré-requis

Voir section 1.2 : travail personnel requis pour le 21 mars.

3.3 Paramètres du programme

1. entrer ou calculer les valeurs adéquates dans le constructeur `Trajectoire()` et dans la méthode `initialiseMatrices()` 2. déterminer le premier instant auquel la trajectoire de la masse coupe théoriquement la barrière lumineuse ; 3. afficher la trajectoire de la masse jusqu'à cet instant.	**F1**

Attention ! Vous avez vu lors du premier TP, que ce programme est sensible aux données, à cause des calculs dans les grands nombres qu'il effectue. Ajustez le rang de calcul des sommes partielles et l'intervalle de définition des fonctions en cas de problèmes numériques.

Ici s'arrête la partie obligatoire du projet.

Dans la suite de l'énoncé, vous sont proposés des exercices facultatifs permettant d'enrichir votre application avec d'autres fonctionnalités.

Premier exercice facultatif (réutilisez les parties de code de `Temperature.java` correspondantes) : 1. enregistrer les données numériques dans un fichier de paramètre `setup.prop` 2. déclarer une variable d'instance de type Properties, à initialiser dans le constructeur par l'appel à la méthode `Properties lectureParametres(String nomFichier)` 3. déclarer deux méthodes génériques de lecture des paramètres : `double getDouble(String nom)` (resp. `int getInteger(String nom)`) pour la lecture de la valeur du paramètre `nom` de type `double` (resp. `int`) 4. initialisez les données de votre programme (dans `Trajectoire()` et `initialiseMatrices()`) au moyen de ces méthodes.	**F2**

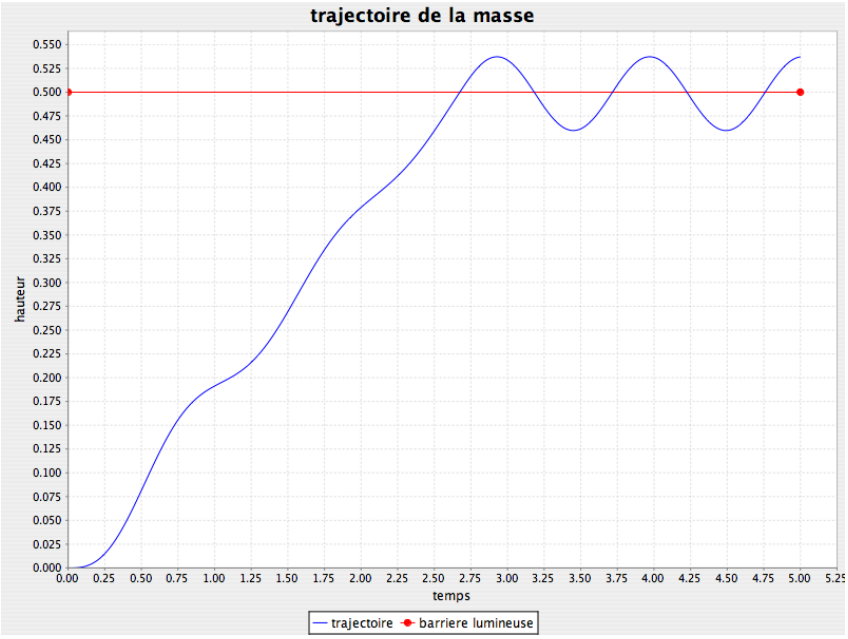
3.4 Trajectoire complète

À l'interruption du lève-charge, à l'instant `tBarriere` où la masse rencontre la barrière lumineuse, la trajectoire de la masse va répondre à une nouvelle équation. Dans un premier temps, en fait, cette équation sera identique à celle de la trajectoire pendant l'élévation de la charge : seule les données d'entrée sont modifiées pour tenir compte de

- l'arrêt du treuil (certains termes des matrices de calcul sont annulés)
- l'état du système au moment de l'arrêt (qui devient alors l'état initial du système)

Second exercice facultatif : affichage de la trajectoire complète.

1. implémenter une méthode `reinitialiseMatrices(Matrice)` qui effectue la réinitialisation adéquate des valeurs des matrices de calcul à partir de l'état du système passé en argument **F3**
2. faire appel à cette méthode dans `ajouteSerieTronquee(TraceSerie)`, avec `this.Y(tBarriere)` en argument



Version finale attendue le lundi 31 mars, 20h AU PLUS TARD.

Déposez sur Campus l'archive complète de votre projet :

`th2_apa_XXX.zip` ou `th2_apa_XXX.tar.gz`

L'archive doit au moins contenir : **G0**

- les deux fichiers sources `PreTrajectoire.java` et `Trajectoire.java`
- le fichier image (`.png`) représentant le graphique résultat
- le fichier de paramètres (si défini, voir **F2**)