```
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class PreTrajectoire {

	// variables d'instances
	private int I;			// rang des sommes partielles
	private double tPas;	// pas d'abscisse  (interpolation)
	private double tMin;	// abscisse minimale
	private double tMax;	// abscisse maximale
	private double xBarriere;	// ordonnee maximale

	// matrices particulieres utilisees pour les calculs
	private Matrice M;	// matrice carree 3x3
	private Matrice M1;	// vecteur 3x1
	private Matrice M2;	// vecteur 3x1
	private Matrice M3;	// vecteur 3x1

	// constructeur
	PreTrajectoire() {
		this.I = 100;
		this.tPas = 0.01;
		this.tMin = 0;
		this.tMax = 5;
		this.xBarriere = 4.5;
		this.initialiseMatrices();
	}

	/**
	 * initialise les matrices particulieres M, M1, M2, M3
	 */
	public void initialiseMatrices() {
		double a = 50;
		double b = 1;
		double[][] exemple0 = {{1,0,0},{0,1,0},{0,0,1}}; // Matrice.identite(3)
		double[][] exemple1 = {{0,1,0},{-a,0,a},{b,0,-b}};
		this.M = new Matrice(exemple1);
		double[][] val1 = {{1},{1},{1}};
		this.M1 = new Matrice(val1);
		this.M2 = new Matrice(val1);
		this.M3 = new Matrice(val1);
	}


	////////////////////////////////////////////////////////////////////////////////
/////////
	/////		PARTIE 1: SOMMES PARTIELLES, COMPLEXITE ET PROBLEMES NUMERIQUES
    /////
	////////////////////////////////////////////////////////////////////////////////
/////////


	/**
	 * calcul de la factorielle de j
	 * @param j entier positif
	 * @return j! entier
	 */
	public int factorielle(int j) {
		int res = 1;
		for (int i = 2; i <= j; i++) {
			res *= i;
		}
		return res;
```



```
	}

	/**
	 * calcul de la somme partielle de terme (i!) pour i=0..I
	 * version lente: appel a la methode factorielle
	 * parametre d'instance: I entier
	 * @return SOMME[0;I](i!) entier
	 */
	public int sommeFactorielleNaif() {
		int somme = 1;
		for (int i = 1; i <= this.I; i++) {
			somme += this.factorielle(i);
		}
		return somme;
	}

	/**
	 * calcul de la somme partielle de terme (i!) pour i=0..I
	 * version rapide: calcul sequentiel des termes de la somme
	 * parametre d'instance: I entier
	 * @return SOMME[0;I](i!) entier long
	 */
	public long sommeFactorielle() {
		long terme = 1;
		long somme = terme;
		for (int i = 1; i <= this.I; i++) {
			terme *= i;
			somme += terme;
		}
		return somme;
	}

	/**
	 * calcul de la somme partielle de terme (t^i) pour i=0..I
	 * PAR CONVENTION, on definit 0^i = lim(t->0) t^i =1 si i=0, 0 sinon
	 * parametre d'instance: I entier
	 * @param t flottant
	 * @return SOMME[0;I](t^i) flottant
	 */
	public double sommeExposant(double t) {
		double terme = 1;
		double somme = 1;
		for (int i = 1; i <= this.I; i++) {
			terme *= t;
			somme += terme;
		}
		return somme;
	}

	/**
	 * calcul de la somme partielle de terme (M^i) pour i=0..I
	 * PAR CONVENTION, on definit 0^i = Identite si i=0, 0 sinon
	 * parametre d'instance: I entier, M Matrice
	 * @return SOMME[0;I](M^i) Matrice
	 */
	public Matrice sommeExposant() {
		Matrice somme = Matrice.identite(3);
		Matrice terme = new Matrice(this.M);
		somme.ajoute(terme);
		for (int i = 2; i <= this.I; i++) {
			terme.multiplieDroite(this.M);
			somme.ajoute(terme);
		}
		return somme;
```

```
	}

	/**
	 * calcul de la somme partielle de terme (M^i.t^i/i!) pour i=0..I
	 * parametre d'instance: I entier, M Matrice
	 * @param t flottant
	 * @return SOMME[0;I](t^i/i!) flottant
	 */
	public Matrice sommeExponentielle(double t) {
		Matrice somme = Matrice.identite(3);
		Matrice terme = new Matrice(this.M);
		terme.multiplie(t);
		somme.ajoute(terme);
		for (int i = 2; i <= this.I; i++) {
			terme.multiplieDroite(this.M);
			terme.multiplie(t/i);
			somme.ajoute(terme);
		}
		return somme;
	}


	////////////////////////////////////////////////////////////////////////////////
	/////                          PARTIE 2: CALCUL DES FONCTIONS
	////////////////////////////////////////////////////////////////////////////////

	/**
	 * calcul de A = BC + Dt + E
	 * ATTENTION: les matrices B, C, D, E ne doivent pas etre modifiees !!!!
	 * @param C,D,E matrice nx1 (n quelconque)
	 * @param B matrice nxn
	 * @param t flottant
	 * @return A matrice nx1
	 */
	public Matrice combinaisonMatrices(Matrice B, Matrice C, Matrice D, Matrice E, double t) {
		Matrice A = new Matrice(E);
		A.ajoute(Matrice.produit(B,C));
		A.ajoute(Matrice.produit(t,D));
		return A;
	}

	/**
	 * calcule la combinaison particuliere Y(t) = exp(t.M).M1 + t.M2 + M3
	 * parametres d'instance: M Matrice 3x3; M1, M2, M3 matrices 3x1
	 * @param t flottant
	 * @return Y(t) matrice 3x1
	 */
	public Matrice Y(double t) {
		return this.combinaisonMatrices(this.sommeExponentielle(t),this.M1,this.M2,this.M3,t);
	}

	/**
	 * x(t) peut etre toute fonction croissante definie sur les reels positifs, par exemple: x(t)=t
	 */
	public double x(double t) {
		//return t;
		return this.Y(t).getElement(0,0);
	}
```



```
	/**
	 * soit x(t) une fonction CROISSANTE sur les reels 0<=t<tMax, et soit une valeur X>x(0)
	 * recherche le plus petit element dans l'image inverse de X par x
	 * retourne une valeur approchee par defaut a tPas pres, si cet element existe; retourne tMax sinon
	 * parametres d'instance: tPas, tMax flottants
	 * @param X flottant
	 * @return min( inf(x^-1(y))-e , tMax) avec |e|<tPas flottant
	 */
	public double xInverse(double X) {
		double t=0;
		boolean stop = false;
		while (!stop && t<this.tMax) {
			t += this.tPas;
				if ( this.x(t) > X) {
					stop = true;
				}
		}
		return t-this.tPas;
	}


	////////////////////////////////////////////////////////////////////////////////
	/////                          PARTIE 3: TRACE DES COURBES
	////////////////////////////////////////////////////////////////////////////////


	/**
	 * cree une serie de N points 2D sur la courbe (t,x(t)) et l'ajoute a l'objet traceur:
	 * la serie est enregistree dans une matrice Nx2 dont la j-eme ligne (0<=j<N) est formee
	 * d'un temps t [colonne 0] et de son image x(t) [colonne 1]
	 * t varie entre tMin et tMax, avec un espacement de tPas, donc: N = (tMax-tMin)/tPas + 1
	 * parametre d'instance: tPas, tMin, tMax flottants; la methode double x(double)
	 * @param
	 * @param traceur               traceur de courbes
	 */
	public void ajouteSerie(TraceSeries traceur) {
		// nombre de points d'interpolation des courbes
		int N = (int)((this.tMax-this.tMin)/this.tPas)+1;
		double[][] serie = new double[N][2];
		double t = tMin;
		for (int i = 0; i < N; i++) {
			serie[i][0] = t;
			serie[i][1] = this.x(t);
			t += this.tPas;
		}
		traceur.ajouteSerieAna(serie, "x(t)");
	}

	/**
	 * cree une serie de N points 2D sur la courbe (t,y(t)) et l'ajoute a l'objet traceur:
	 * t varie entre tMin et tMax, avec un espacement de tPas, donc: N == (tMax-tMin)/tPas + 1
	 * x(t) etant croissante, soit tBarriere = min { t | x(t) >= xBarriere },
	 * la courbe t -> y(t) est la fonction lineaire par morceaux definie par:
```

```
	 * y(t) = x(t) tant que t <= tBarriere, y(t) est constante =x(tBarriere) ensuite
	 * parametre d'instance: tPas, tMin, tMax, xBarriere flottants; la methode double x(
double)
	 * @param traceur               traceur de courbes
	 */
	public void ajouteSerieTronquee(TraceSeries traceur) {
		// nombre de points d'interpolation des courbes
		int N = (int)((this.tMax-this.tMin)/this.tPas)+1;
		double[][] serie = new double[N][2];
		double t = tMin;
		int i = 0;
		do {
			serie[i][0] = t;
			serie[i][1] = this.x(t);
			t += this.tPas;
			i++;
		} while (serie[i-1][1]<=this.xBarriere && i<N);
		double tBarriere = serie[i-1][0];
		System.out.println("tBarriere: " + tBarriere);
		double xConstante = serie[i-1][1];
		while (i<N) {
			serie[i][0] = t;
			serie[i][1] = xConstante;
			t += this.tPas;
			i++;
		}
		traceur.ajouteSerieExpe(serie, "x tronquee (t="+tBarriere+")");
	}

	/**
	 * cree et affiche un graphique de la courbe (t,x(t))
	 */
	public void traceCourbe () {
		TraceSeries traceur = new TraceSeries();
		this.ajouteSerie(traceur);
		this.ajouteSerieTronquee(traceur);
		traceur.creeGraphique("courbe x", "t", "x(t)");
		traceur.afficheGraphique("courbe t -> x(t)");
		traceur.enregistreGraphique("courbe_x");
	}

	////////////////////////////////////////////////////////////////////////////////
////////
	////////////////////////////////////////////////////////////////////////////////
////////
	/////                              MAIN: VERIFICATION ET TEST
    /////
	////////////////////////////////////////////////////////////////////////////////
////////
	////////////////////////////////////////////////////////////////////////////////
////////

	/**
	 * main:
	 * @param args liste des arguments: nom du fichier de proprietes
	 */
	public static void main (String[] args){
		PreTrajectoire T = new PreTrajectoire();

		//////////////////////////////////////////////////////////////////////////
////////////////
		// Q1: implementer factorielle(int)
		//     verifier: T.factorielle(0)  == 1
		//     verifier: T.factorielle(10) == 3628800
```



```
		//     combien d'operations (multiplication d'entiers) sont effectuees par l
a methode factorielle(10) ?
		//     combien d'operations (multiplication d'entiers) sont effectuees par l
a methode factorielle(j), j>1 ?
		System.out.println("facto(0):" + T.factorielle(0));
		System.out.println("facto(10):" + T.factorielle(10));

		//////////////////////////////////////////////////////////////////////////
////////////////
		// Q2: implementer sommeFactorielleNaif()
		//     (T.I=10) verifier: T.sommeFactorielleNaif() == 4037914
		//     combien d'operations (somme et multiplication d'entiers) sont effectu
ees par sommeFactorielleNaif() pour I=10 ?
		System.out.println("sommeFactoNaif: " + T.sommeFactorielleNaif());

		//////////////////////////////////////////////////////////////////////////
////////////////
		// Q3: implementer sommeFactorielle()
		//     (T.I=10) verifier: T.sommeFactorielle() == 4037914
		//     combien d'operations (somme et multiplication d'entiers) sont effectu
ees par sommeFactorielle() pour I=10 ?
		System.out.println("sommeFacto: " + T.sommeFactorielle());

		//////////////////////////////////////////////////////////////////////////
////////////////
		// Q3a: poser I=100 dans le constructeur PreTrajectoire()
		//     verifier: T.sommeFactorielle() == 1005876315485501978
		//     expliquer pourquoi T.sommeFactorielleNaif() ne retourne pas ce result
at ?
		//     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)  == 2.5353012004564588E30 > 2.10^30
		//     verifier: T.sommeExposant(10) == 1.1111111111111118E100 > 1.10^100, e
xpliquer ce chiffre 8 ?
		//     calculer: T.sommeExposant(10000)
		System.out.println("sommeExp(0): " + T.sommeExposant(0));
		System.out.println("sommeExp(1): " + T.sommeExposant(1));
		System.out.println("sommeExp(2): " + T.sommeExposant(2));
		System.out.println("sommeExp(10): " + T.sommeExposant(10));
		System.out.println("sommeExp(10000): " + T.sommeExposant(10000)); // Infinit
y

		//////////////////////////////////////////////////////////////////////////
////////////////
		// Q4a: 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 taill
e 3: Id3)
		//     verifier: T.sommeExposant() == 501.Id3
		System.out.println("sommeExp():\n" + T.sommeExposant());

		//////////////////////////////////////////////////////////////////////////
////////////////
		// Q5a: (T.I=500) poser M={{0,1,0},{-50,0,50},{0,0,0}} dans initialiseMatric
```

```
es() : a=50, M=new Matrice(exemple1)
                //      calculer T.sommeExposant(), interpreter

                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // Q5b: (T.I=500) poser M={{0,1,0},{0,0,0},{0,0,0}} dans initialiseMatrices(
) : a=0
                //      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
                System.out.println("expo(M,0):\n" + T.sommeExponentielle(0));
                System.out.println("expo(M,2):\n" + T.sommeExponentielle(2));

                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // Q6a: (T.I=500) poser M=Id3 dans initialiseMatrices()
                //      verifier: T.sommeExponentielle(0) == Id3
                //      verifier: T.sommeExponentielle(2) == (7.389056098930649).Id3

                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // Q7: implementer combinaisonMatrices(Matrice,Matrice,Matrice,Matrice,doubl
e)
                //      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,0) == 2.C
                //      verifier: T.combinaisonMatrices(B,C,C,C,2) == 4.C
                Matrice B = Matrice.identite(3);
                double[][] vals = {{1},{1},{1}};
                Matrice C = new Matrice(vals);
                System.out.println("combi(0):\n" + T.combinaisonMatrices(B,C,C,C,0));
                System.out.println("combi(2):\n" + T.combinaisonMatrices(B,C,C,C,2));

                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // 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
                System.out.println("Y(Id3,0):\n" + T.Y(0));
                System.out.println("Y(Id3,2):\n" + T.Y(2));


                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // Q9: implementer ajouteSerie(TraceSeries) [voir TemperatureAnalytique.java
]
                //      implementer traceCourbe()            [voir Temperature.java]
                //      afficher la courbe x(t)
                T.traceCourbe();

                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // Q10: implementer ajouteSerieTronquee(TraceSeries)
                //      modifier traceCourbe() pour afficher la courbe tronquee
                //      faire afficher tBarriere par ajouteSerieTronquee(TraceSeries)

                ////////////////////////////////////////////////////////////////////////////////
////////////////
```



```
                // Q10a: modifier x(double) de sorte que x(t) == le premier element du vecte
ur Y(t) pour tout t
                //      poser I=100, M={{0,1,0},{-50,0,50},{1,0,-1}}

                ////////////////////////////////////////////////////////////////////////////////
////////////////
                // questions subsidiaires:
                // Q11 interpreter l'allure de la courbe pour tMax=6
                // Q12 modifier ajouteSerieTronquee(TraceSeries), ou plutot, recopier la en
une nouvelle méthode, afin qu'elle saisisse
                //      les TROIS series de points, correspondant aux trois coordonnees du ve
cteur Y(t).

        }
}
```

```
/** Projet integrateur CB1 2008/2009 [ CORRECTION ] */
public class SommesEtMatrices {

	/**
	 * calcul de la factorielle de n
	 * @param n entier positif
	 * @return n! entier
	 */
	public static int factorielle(int n) {
		int res = 1;
		for (int i = 2; i <= n; i++) {
			res *= i;
		}
		return res;
	}
	//Q1 combien d'operations effectuees par 'factorielle(10)' ? // 10
	// quelle est la complexite de factorielle(n) ? // O(n)


	/**
	 * calcul de la somme partielle de terme (n!) pour n=0..rang
	 * version lente: appel a la methode factorielle
	 * @param rang entier positif
	 * @return SOMME[0;rang](n!) entier
	 */
	public static int sommeFactorielleNaif(int rang) {
		int somme = 1;
		for (int n = 1; n <= rang; n++) {
			somme += SommesEtMatrices.factorielle(n);
		}
		return somme;
	}
	//Q2 combien d'operations effectuees par 'sommeFactorielleNaif(10)' ?
	// REPONSE: 1+2+3+...+10 = (10+1)x10/2 = 55
	// quelle est la complexite de sommeFactorielleNaif(n) ? // O(n^2)

	/**
	 * calcul de la somme partielle de terme (n!) pour n=0..rang
	 * version rapide: calcul sequentiel des termes de la somme
	 * @param rang entier positif
	 * @return SOMME[0;rang](n!) entier long
	 */
	public static long sommeFactorielle(int rang) {
		long terme = 1;
		long somme = terme;
		for (int n = 1; n <= rang; n++) {
			terme *= n;
			somme += terme;
		}
		return somme;
	}
	//Q3 combien d'operations effectuees par 'sommeFactorielle(10)' ? // 2x10 = 20
	// quelle est la complexite de sommeFactorielle(n) ? // O(n)
	//Q4 pourquoi sommeFactorielleNaif(100) != sommeFactorielle(100) ?
	// la premiere retourne un int, la seconde un long et MAX_INT < 100! < MAX_LONG

	/**
	 * calcul de la somme partielle de terme (t^n) pour n=0..rang
	 * PAR CONVENTION, on definit 0^n = lim(t->0) t^n =1 si n=0, 0 sinon
	 * @param t flottant, rang entier positif
	 * @return SOMME[0;rang](t^n) flottant
	 */
	public static double sommeExposant(double t, int rang) {
		double terme = 1;
		double somme = 1;
		for (int n = 1; n <= rang; n++) {
			terme *= t;
			somme += terme;
```



```
		}
		return somme;
	}
	//Q5 que dire de sommeExposant(10,500) ? // Somme_{n=1..500} t^n > MAX_DOUBLE

	/**
	 * calcul de la somme partielle de terme (M^n) pour n=0..rang
	 * PAR CONVENTION, on definit 0^n = Identite si n=0, 0 sinon
	 * @param M Matrice d'entiers 3x3, rang entier positif
	 * @return SOMME[0;rang](M^n) Matrice
	 */
	public static Matrice sommeExposant(Matrice M, int rang) {
		Matrice somme = Matrice.identite(3);
		Matrice terme = new Matrice(M);
		somme.ajoute(terme);
		for (int n = 2; n <= rang; n++) {
			terme.multiplieDroite(M);
			somme.ajoute(terme);
		}
		return somme;
	}

	/**
	 * calcul de la somme partielle de terme (M^n.t^n/n!) pour n=0..rang
	 * @param t flottant, M Matrice d'entiers 3x3, rang entier positif
	 * @return SOMME[0;rang](M^n.t^n/n!) Matrice
	 */
	public static Matrice exponentielle(double t, Matrice M, int rang) {
		Matrice somme = Matrice.identite(3);
		Matrice terme = Matrice.identite(3);
		for (int n = 1; n <= rang; n++) {
			terme.multiplieDroite(M);
			terme.multiplie(t/n);
			somme.ajoute(terme);
		}
		return somme;
	}

	/**
	 * calcul de BC + Dt + E
	 * ATTENTION: les matrices B, C, D, E ne doivent pas etre modifiees !!!!
	 * @param C,D,E matrice nx1 (n quelconque), B matrice nxn, t flottant
	 * @return BC + Dt + E matrice nx1
	 */
	public static Matrice combinaisonMatrices(Matrice B, Matrice C, Matrice D, Matric
e E, double t) {
		Matrice A = new Matrice(E);
		A.ajoute(Matrice.produit(B,C));
		A.ajoute(Matrice.produit(t,D));
		return A;
	}

	/**
	 * calcule la combinaison particuliere exp(t.B).C + t.D + E
	 * ATTENTION: les matrices B, C, D, E ne doivent pas etre modifiees !!!!
	 * @param C,D,E matrice nx1 (n quelconque), B matrice nxn, t flottant
	 * @param rang entier (rang de la somme pour le calcul de l'exponentielle)
	 * @return exp(t.B).C + t.D + E matrice nx1
	 */
	public static Matrice combinaisonExponentielleMatrices(Matrice B, Matrice C, Matr
ice D, Matrice E, double t, int rang) {
		return  SommesEtMatrices.combinaisonMatrices(SommesEtMatrices.exponentiel
le(t,B,rang),C,D,E,t);
	}

}
```

```
import java.io.IOException;
import java.awt.*;

/**
 * Projet integrateur CB1 2008/2009
 * ================================================
 * Fonctions de calcul et de trace de l'elevation theorique d'une charge
 * @author sofdem [ CORRECTION ]
 */
public class TrajectoireTheo {

        // constantes d'instances
        private final double RAIDEUR = 5.45;
// raideur du ressort (g/s^2)
        private final double VITESSE_ANGULAIRE_VIDE = 3;
// vitesse de rotation de la poulie a vide (1/s)
        private final double MASSE_MAX = 0.25;
// masse pour l'immobilisation du treuil (g)
        private final double RAYON = 0.025;
// rayon de la poulie (m)
        private final double LONGUEUR_RESSORT_VIDE = 0.03;
// longueur du ressort a vide (m)
        private final double PESANTEUR = 9.81;
// acceleration de la pesanteur (m/s^2)
        private final double MASSE = 0.15;
// masse suspendue (g)
        private final double X_BARRIERE = 0.5;
// position barriere lumineuse (m)

        private final int RANG = 100;           // rang de calcul des sommes partielles
        private final double TPAS = 0.01;       // pas de temps
        private final double TMIN = 0;          // temps minimal
        private final double TMAX = 5;          // temps maximal

        // matrices particulieres utilisees pour les calculs
        private Matrice M;              // matrice carree 3x3
        private Matrice Y0;             // vecteur 3x1  solution au temps t=0
        private Matrice Ysp;            // vecteur 3x1  solution particuliere: pente
        private Matrice Ysp0;           // vecteur 3x1  solution particuliere: terme cons
tant

        private double tBarriere;       // instant d'arret du treuil




        ////////////////////////////////////////////////////////////////////
        /////          PARTIE 1: INITIALISATION DES DONNEES             /////
        ////////////////////////////////////////////////////////////////////

        // constructeur
        TrajectoireTheo() {
                this.tBarriere = 0;
                this.initialiserMatricesPhaseI();
        }

        /**
         * initialise les matrices particulieres M, Y0, Ysp, Ysp0
         */
        public void initialiserMatricesPhaseI() {
                //// termes de la matrice M du systeme
                double w0carre = RAIDEUR/MASSE;
                double wv = VITESSE_ANGULAIRE_VIDE*2*Math.PI;
// wv: vitesse de rotation poulie a vide (m)
                double alpha = wv/(MASSE_MAX*PESANTEUR*RAYON);
// alpha: valeur caracteristique du treuil
                double beta = alpha*RAIDEUR*RAYON*RAYON;
```



```
                //// termes de la solution particuliere Ysp(t)=(Ysp.t+Ysp0)
                double Xv = LONGUEUR_RESSORT_VIDE;

// Xv: position point d'attache a vide (m)
                double X0 = (MASSE*PESANTEUR/RAIDEUR) + Xv;
// X0: position point d'attache a t=0 (m)
                double terme = RAYON*wv + beta*(Xv-X0);

                //// initialisation des matrices
                double[][] matrice = {{0,1,0},{-w0carre,0,w0carre},{beta,0,-beta}};
                this.M = new Matrice(matrice);
                double[][] ini = {{0},{0},{X0}};
                this.Y0 = new Matrice(ini);
                double[][] sol1 = {{terme},{0},{terme}};
                this.Ysp = new Matrice(sol1);
                double[][] sol0 = {{-X0},{terme},{0}};
                this.Ysp0 = new Matrice(sol0);
        }

        /**
         * reinitialise les matrices M, Y0, Ysp, Ysp0 en fonction d'une matrice Y1
         */
        public void initialiserMatricesPhaseII(Matrice Y1) {
                //// reinitialisation: les termes 'beta' et 'terme' s'annulent
                this.M.setElement(2,0,0);
                this.M.setElement(2,2,0);
                this.Ysp.setElement(0,0,0);
                this.Ysp.setElement(2,0,0);
                this.Ysp0.setElement(1,0,0);
                //// Y1 est la solution au nouveau temps initial
                this.Y0 = new Matrice(Y1);
        }




        ////////////////////////////////////////////////////////////////////
        /////             PARTIE 2: CALCUL DES FONCTIONS               /////
        ////////////////////////////////////////////////////////////////////

        /**
         * calcule et retourne Y(t) = exp(t.M) (Y0-Ysp0) + t.Ysp + Ysp0
         * @param t flottant
         * @return Y(t) matrice 3x1
         */
        public Matrice Y(double t) {
                return SommesEtMatrices.combinaisonExponentielleMatrices (this.M,
               Matrice.soustrait(this.Y0,Ysp0), this.Ysp, this.Ysp0, t, RANG);
        }

        /**
         * retourne le premier element du vecteur Y(t) (fonction croissante)
         * @param t flottant
         * @return x(t) flottant
         */
        public double x(double t) {
                return this.Y(t).getElement(0,0);
        }
```

```
        ////////////////////////////////////////////////////////////////
        /////               PARTIE 3: TRACE DES COURBES              /////
        ////////////////////////////////////////////////////////////////

        /**
         * 1) retourne une courbe interpolant les points 2D (t,x(t)) pour
         * t variant entre TMIN et TMAX avec un espacement de TPAS
         * 2) affiche la valeur de tBarriere = min { t | x(t) >= X_BARRIERE },
         * parametre d'instance: TPAS, TMIN, TMAX, X_BARRIERE flottants
         */
        public Courbe creeCourbe() {
                Courbe c = new Courbe("phase I+II");
                c.setCouleur(Color.getHSBColor(0.75f, .9f, .9f));
                double t = TMIN;
                double lastX = this.x(t);

                while (lastX < X_BARRIERE && t <= TMAX) {
                        c.ajouter(t, lastX);
                        t += TPAS;
                        lastX = this.x(t);
                }

                this.tBarriere = t;
                System.out.println("tBarriere: " + this.tBarriere);
                this.initialiserMatricesPhaseII(this.Y(this.tBarriere));
                while (t<=TMAX) {
                        c.ajouter(t, this.x(t-this.tBarriere));
                        t += TPAS;
                }
                return c;
        }

        public Courbe creeCourbeBarriere() {
                Courbe c = new Courbe("tBarriere = " + this.tBarriere, 240);
                c.setCouleur(Color.getHSBColor(.9f, .9f, .75f));
                c.ajouter(this.tBarriere-1E-7, 0);
                c.ajouter(this.tBarriere, X_BARRIERE);
                c.ajouter(this.tBarriere+1E-7, 0.6);
                return c;
        }

        public Courbe creeCourbeXBarriere() {
                Courbe c = new Courbe("xBarriere = " + X_BARRIERE);
                c.setCouleur(Color.getHSBColor(.9f, .9f, .75f));
                for (double t = TMIN; t <= TMAX; t += TPAS) {
                c.ajouter(t, X_BARRIERE);
                }
                return c;
        }

        /**
         * main: cree un objet Graphique et affiche la courbe
         * representant la trajectoire theorique de la courbe
         * @param args liste des arguments
         */
        public static void main (String[] args) {
                Graphique g = new Graphique( "Courbe t -> x(t)", 800, 600);
                g.setMaxY(0.6);
                TrajectoireTheo theo = new TrajectoireTheo();
                g.ajouter(theo.creeCourbe());
                g.ajouter(theo.creeCourbeBarriere());
                g.ajouter(theo.creeCourbeXBarriere());
                g.montrer();
        }

}
```

```
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.awt.*;

/**
 * Projet integrateur CB1 2008/2009
 * ================================================
 * Fonctions de lecture et de trace de l'elevation experimentale d'une charge
 * @author sofdem [ CORRECTION ]
 */
public class TrajectoireExpe {

        // variables d'instance
        private String nomFichier;                      // nom du fichier
        private Courbe courbeExpe;                      // courbe experimentale

        // constructeur
        TrajectoireExpe() {
                this.nomFichier = "resultatsexpe.txt";
                this.courbeExpe = new Courbe("expe", 53);
                this.courbeExpe.setCouleur(Color.getHSBColor(.5f, .9f, .5f));
                this.lectureFichier();
        }

        // accesseurs
        public Courbe getCourbe() { return this.courbeExpe; }

        /**
         * lecture du fichier texte de donnees formate en
         * 3 colonnes separees par des tabulation/espaces : t   non-lu   x(t)
         * initialisation de courbeExpe
         */
        private void lectureFichier () {
                try {
                        BufferedReader f = new BufferedReader(new FileReader(this.nomFich
ier));
                        String ligne = f.readLine();
                        while (ligne != null) {
                                String[] tabLigne = ligne.split("[ \t]+");
                                double t = Double.parseDouble(tabLigne[0].replace(',','.'
));
                                double x = Double.parseDouble(tabLigne[2].replace(',','.'
));
                                this.courbeExpe.ajouter(t,x);
                                ligne = f.readLine();
                        }
                } catch (IOException e) {
                        System.out.println("erreur dans la lecture du fichier" + this.nom
Fichier);
                }
        }

        /**
         * main: affiche les courbes experimentales et theoriques
         * @param args liste des arguments
         */
        public static void main (String[] args) {
                Graphique g = new Graphique( "Courbe t -> x(t)", 800, 600);
                TrajectoireExpe expe = new TrajectoireExpe();
                g.ajouter(expe.getCourbe());
                TrajectoireTheo theo = new TrajectoireTheo();
                g.ajouter(theo.creeCourbe());
                g.ajouter(theo.creeCourbeXBarriere());
                g.montrer();
        }
}
```