Bac 1 en sciences de l'ingénieur
Université de Liège
Année académique 2016/2017

Projet d'introduction à l'analyse numérique : Stabilité d'un pont suspendu soumis à des sollicitations cycliques

Travail réalisé par : Jean Michel Muramutsa, Johan Wandji Koe, Clément Waleffe

Question 1 : Algorithmes de la sécante et de la bissection et interpolation :

L'objectif de cette première question peut se décomposer en deux parties. La première partie concerna l'interpolation, tandis que la deuxième sera centrée sur la méthode de la sécante et de la bissection.

1.1 Interpolation :

Il s'agira ici de créer une fonction représentant l'évolution de l'enveloppe de l'angle de torsion en fonction du temps à partir d'un tableau Excel. Celui-ci comportant certaines valeurs d'angle (en radian) pour en certain temps.

Pour commencer, nous utilisons la fonction interne à MATLAB : xlsread. Celle-ci forme un tableau MATLAB (2 colonnes et 71 lignes) à partir de fichier Excel. Cela nous permettra de créer un vecteur 'temps' et un vecteur 'angles' comportant les données du tableau Excel.

Avant de commencer l'interpolation, il est nécessaire de définir un intervalle de temps et un pas. L'intervalle de temps ira de 0 à 300 de manière à englober toutes les valeurs. Le pas détermine, lui, de quelle manière l'intervalle va être décomposé. Plus le pas est petit, plus l'intervalle donné contient de valeur et plus l'interpolation se fera avec précision. Nous avons jugé qu'il n'était pas nécessaire d'utiliser un pas plus petit que 10^{-4} car celui permet déjà une très bonne précision.

Nous pouvons dès lors commencer l'interpolation. Il existe 3 méthodes différentes dont nous allons discuter pour réaliser ce processus.

1.1.1) Interpolation polynomiale :

La première méthode est l'interpolation polynomial. Cette méthode consiste à créer un unique polynôme passant par tous les points à disposition. Nous cherchons donc un polynôme $P(x) = \sum_{i=0}^{n-1}(a_i * x^i)$. Il existe une solution unique pour ce polynôme si et seulement si tous les x_i sont différents. Par n points d'un plan, on peut faire passer un polynôme de degré n-1. Dans notre cas, le degré maximal du polynôme est 70 mais nous allons voir que ce n'est pas le degré le plus élevé qui donne le résultat le plus fidèle. Pour réaliser cette méthode, nous faisons appelle aux fonctions *polyfit(x, y, n)* et *polyval(p, x)* de MATLAB. La première retourne les coefficient d'un polynôme $P(x)$ de degré n qui réalise la meilleure approximation des points formés par les vecteurs x et y entrés en argument. La deuxième retourne la valeur d'un polynôme (caractérisé par le vecteur p) évalué en x.

Comment nous pouvons le constater, la représentation de la fonction par un polynôme de degré 5 (*figure 1.1*) est très loin du résultat attendu. Néanmoins, en montant dans les degrés, nous apercevons que la méthode polynomiale se rapproche de plus en plus du résultat attendu. Le degré 15 (*figure 1.2*) reste encore trop imprécis à nos yeux et le degré 20 (*figure 1.3*) semble être une bonne approximation. Cependant, nous pouvons déjà remarquer le début de fortes oscillations au temps 0 ainsi qu'au temps 300. Plus les degrés du polynôme vont augmenter, plus ces oscillations vont devenir forte comme constaté sur le zoom de la polynomial de degré 25 (*figure 1.4*). C'est pourquoi nous décidons d'utiliser le polynôme de degré 20 pour cette méthode.

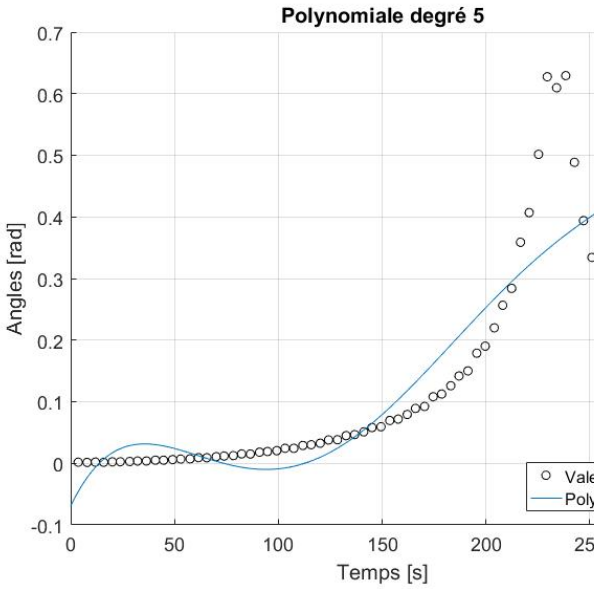


Figure 1.1 : Polynomiale de degré 5

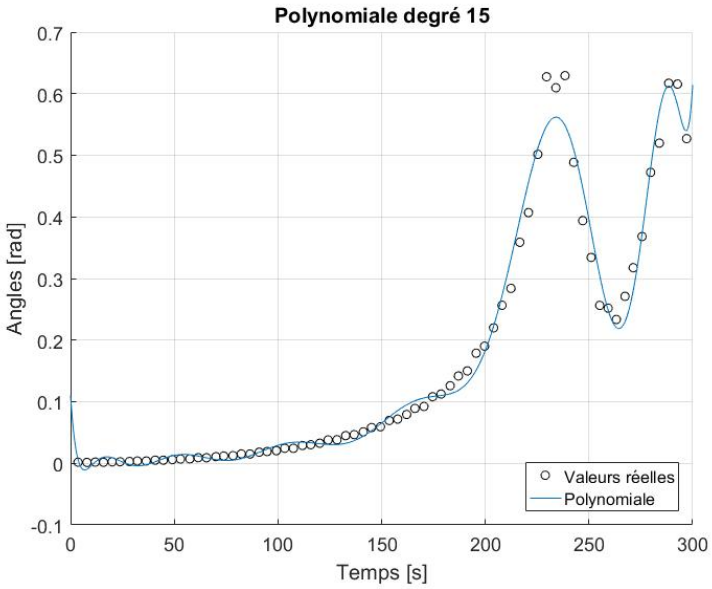


Figure 1.2 : Polynomiale de degré 1

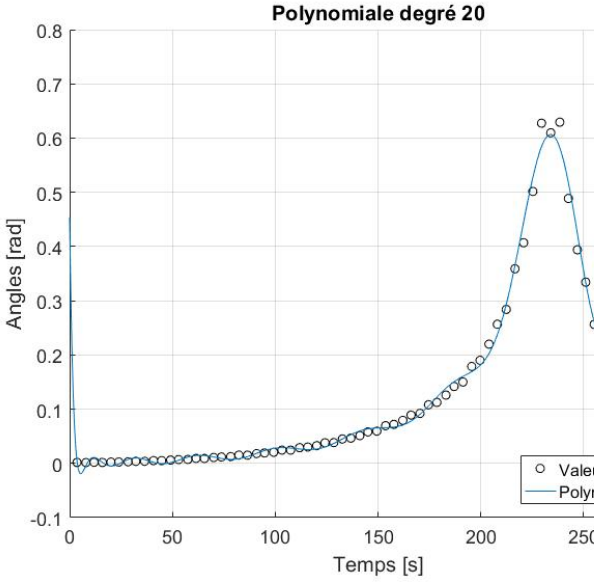


Figure 1.3 : Polynomiale de degré 20

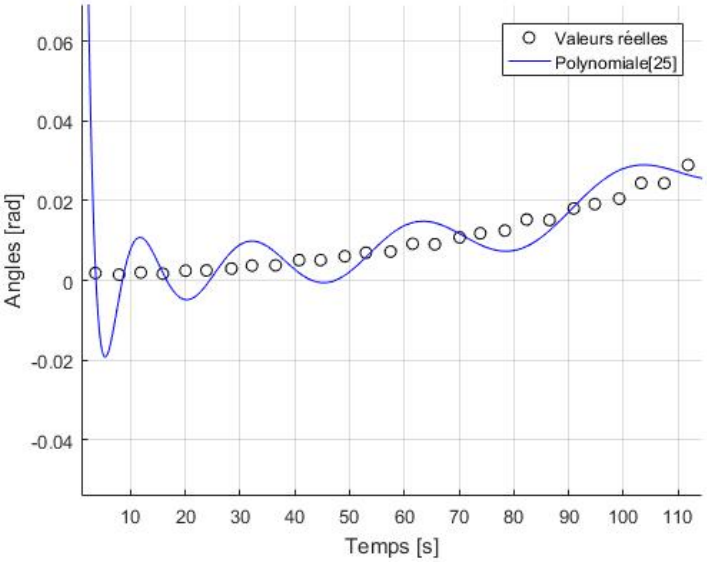


Figure 1.4 : zoom sur la polynomiale de degré 25

1.1.2 Interpolation linéaire :

La méthode de l'interpolation linéaire va simplement relié tous les points entre eux de manière linéaire en utilisant pour cela un polynôme de degré 1 entre chaque point du plan. Cette méthode est invoquée via *interp1(x, y, xq)* qui retourne une approximation linéaire à partir de 2 vecteurs x et y, et un vecteur xq représentant la variation de x sur l'intervalle avec un certain pas.

1.1.3 Interpolation par splines cubique :

Pour terminer, nous avons l'interpolation par splines cubique. Cette méthode est basée sur une « interpolation par morceaux ». Elle utilise également le processus d'approximation par polynôme sauf que dans ce cas-ci, ce mécanisme est réalisé entre chaque paire de points. Nous avons donc un polynôme différent entre chaque point et de degré variable. Nous invoquons cette méthode dans MATLAB grâce à *interp1(x,y,xq,'spline')* où les données en argument de la fonction sont les mêmes que pour l'interpolation linéaire et où *'spline'* définit simplement la méthode appliquée.

Cette méthode est visiblement la plus fidèle et nous l'utiliserons donc pour la suite.

1.2 Méthode de la bissection et de la sécante :

L'objectif ici est de trouver la valeur exacte du temps nécessaire pour atteindre une amplitude de l'angle de torsion donné (0.3 radians pour l'exercice). De manière général, il faut créer deux fonctions capables de déterminer les racines d'une fonction MATLAB $f(x)$ à partir de deux points initiaux : x_0 et x_1. Nous ferons l'hypothèse que $f(x)$ est continue et que $x_0 < x_1$. Nous introduisons ici, une tolérance qui détermine l'erreur admise pour une certaine valeur recherchée. Ainsi qu'un nombre d'itération maximal à ne pas dépasser au cas où il y aurait un problème avec notre code, ou avec les arguments de la fonction (par exemple une fonction qui n'est pas continue sur l'intervalle).

1.2.1 Bissection :

Pour cette méthode, nous devons rajouter l'hypothèse que $f(x_0)$ et $f(x_1)$ se trouve d'une part et d'autre de la racine. Ainsi, par le théorème des valeurs intermédiaires, nous savons qu'il existe une racine sur l'intervalle $[x_0, x_1]$. Dès lors, le principe est le suivant : une boucle dans la fonction va tourner tant que l'erreur entre la valeur recherchée et l'évaluation de la fonction en x_0 est plus grande que la tolérance. A chaque itération la fonction va, d'une part, calculer le milieu de l'intervalle $[x_0, x_1]$

$$x_i = \frac{(x_{0(i-1)} * x_{1(i-1)})}{2}$$

D'autre part, la fonction va analyser la valeur de f(milieu) et déduire si cette valeur est inférieure ou supérieure à la racine. En faisant l'hypothèse que $f(x_0) < f(x_1)$, si f(milieu) < *valeur recherchée* (cas 1), $x_{0(i)}$ devient le milieu de l'intervalle et $x_{1(i)}$ ne varie pas. Si f(milieu) < *valeur recherchée* (cas 2), c'est $x_{1(i)}$ qui devient le milieu et $x_{0(i)}$ qui ne varie pas. Ainsi, l'intervalle contenant la racine va être divisé par 2 à chaque itération et la racine recherchée sera de plus en plus « à l'étroit » dans cet intervalle. Dès que l'erreur devient plus petite que la tolérance, la boucle s'arrête et la bissection sort la dernière valeur du milieu. Dans le cas où le nombre d'itération de la boucle dépasserait le nombre maximal d'itération (ce qui serait anormal), la boucle stoppe et la fonction renvoie un message d'erreur.
Nb : 1) Dans le cas où $f(x_0) > f(x_1)$, le processus reste le même et les cas 1 et 2 sont inversés.

2) $x_0 = x_{0\,(0)}$ et $x_1 = x_{1\,(0)}$

1.2.2 Sécante :

Les hypothèses concernant les valeurs de $f(x_0)$ et $f(x_1)$ n'ont plus lieu d'être pour cette méthode. Le principe de boucle est le même que pour la bissection, cependant son contenu n'est pas le même. Ici, à chaque itération la fonction calcul une racine x d'une approximation de la fonction. Cette approximation est la droite qui relie le point $(x_{i-1}, f(x_{i-1}))$ et le point $(x_i, f(x_i))$ en sachant qu'initialement $x_{i-1} = x_0$ et $x_i = x_1$.

Nous avons donc,

$$x_{i+1} = x_i - \frac{f(x_i) * (x_i * x_{i-1})}{f(x_i) - f(x_{i-1})}$$

Ensuite, x_{i-1} devient x_i, x_i devient x_{i+1} et l'opération est répétée jusqu'à avoir une racine de l'approximation assez proche de la racine de la fonction initiale. A nouveau, si le nombre d'itération maximale est dépassée, la fonction s'arrête et renvoie un message d'erreur.

1.2.3 Résultats et comparaison des deux méthodes :

Pour une amplitude de 0.3 rad et une tolérance de 10^{-5} nous obtenons un temps de 213.4514s avec la sécante et 213.4512s avec la bissection. Comme nous pouvons le voir sur la *figure 1.5*, la sécante est légèrement plus précise (point bleu). De plus cette méthode est plus rapide (environ 1.5 secondes) et effectue moins d'itération pour arriver au résultat (5 itérations) que la bissection (environ 6 secondes et 10 itérations).

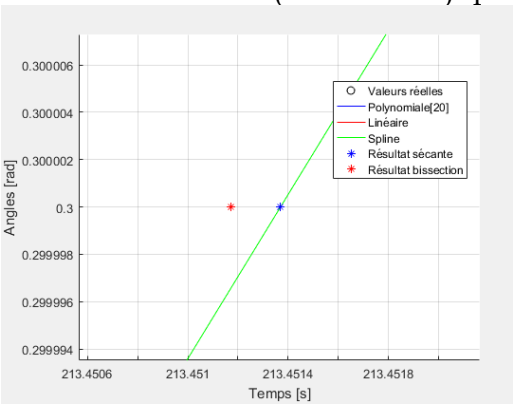


Figure 1.5: zoom sur le résultat de la sécante et de la bissection

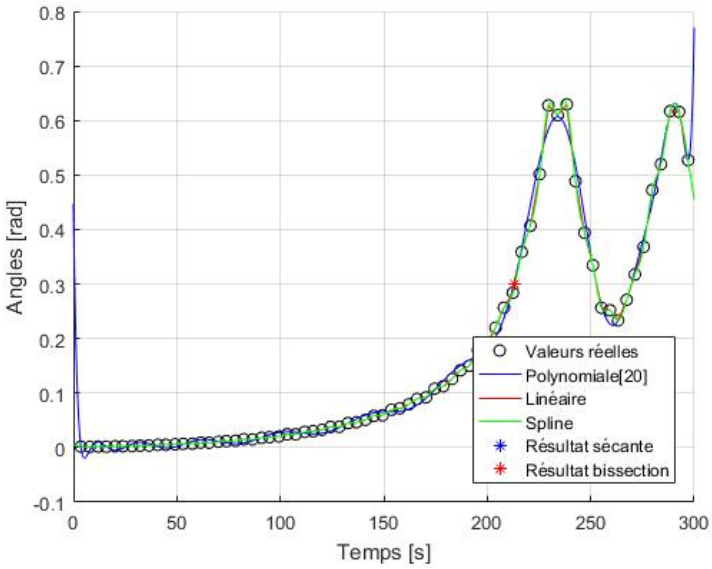


Figure 1.6: Vue globale de la question 1

En prenant introduisant la fonction cosinus dans les deux méthodes (tolérance de 10^{-5}, $x_0 = 1$, $x_1 = 3$, $valeur\ recherchée = 0$), nous observons à nouveau une différence pour le nombre d'itération et temps de travail : 17 itérations et 0.003 secondes pour la bissection alors que seulement 5 itérations et 0.001 secondes pour la sécante.

Nous aurons donc une préférence pour la sécante si nous devons choisir entre les deux méthodes. La figure 1.6 récapitule la question 1 en exposant sur le même graphique les différentes méthodes d'interpolation, ainsi que le résultat de la sécante et de la bissection.

Question 2 : Mise en place de la modélisation:

Il est question ici d'utiliser les méthodes numériques permettant d'illustrer la simulation du système du pont suspendu. En exploitant la méthode de *Euler explicite* et la fonction *ode45*, un dérivé des méthodes de *Runge-Kunta*, nous ébaucherons l'évolution de y , l'élongation du tablier et de de son angle avec l'horizontale θ.

2.1. Systèmes d'équations différentielles :

Tout d'abord, il est essentiel de remanier les équations du mouvement initialement données :

$$my'' = -c_y y' - [f(y^-) + f(y^+)] + Asin(\omega t)$$

$$\frac{ml^2}{3}\theta'' = -c_\theta \theta' + l[f(y^-) - f(y^+)]cos\theta$$

avec y^+ = y + l*sinθ et y^-= y – l*sinθ

En effet, ce système peut-être transformé en un système d'équations différentielles d'ordre 1 plus intuitif à résoudre via les méthodes numériques. Cependant, le comportement des câbles conditionne les calculs, et dons le système. Il y aura donc 2 modèles : le modèle linéaire, dont la force appliquée sur les cordes s'exprime comme $f_{lin} = Kx$; et le modèle non-linéaire, dont $f_{nl} = \frac{K}{\alpha}(e^{\alpha x} - 1)$

2.2. Calcul des dérivées :

Nous créons les fonctions *odefunction_lin* et *odefunction_nl* ayant en paramètre le vecteur temps t et le vecteur y de composantes $[y, y', \theta, \theta']$. Nous avons choisi de traduire le problème vectoriellement par souci de facilité et surtout de rapidité d'exécution du programme.

Les dérivées sont calculées en créant une matrice M telle que :

```
M_lin=zeros(4,1);
M_lin(1) = y(2);
M_lin(2) = (1/m)*(-(cy*y(2))-(2*K*y(1))+(A*sin(omega*t)));
M_lin(3) = y(4);
M_lin(4) = (3/(m*(l^2)))*(-(ctheta*y(4))-(2*K*(l^2)*sin(y(3))*cos(y(3))));
```

Il en ressort un vecteur M = $[y', y'', \theta', \theta'']$.

2.3. Méthode de Euler Explicite :

La méthode de Euler explicite, soit la méthode de Taylor à l'ordre 1, repose sur la formule

$$x_{i+1} = x_i + hf(x_i, t_i)$$

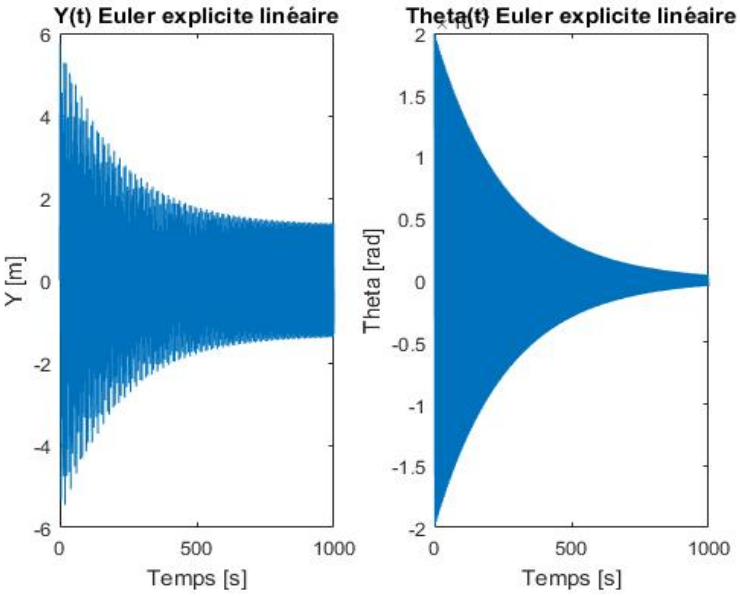
avec $f(x_i, t_i) = x'(t_i)$ et h, le pas de temps.

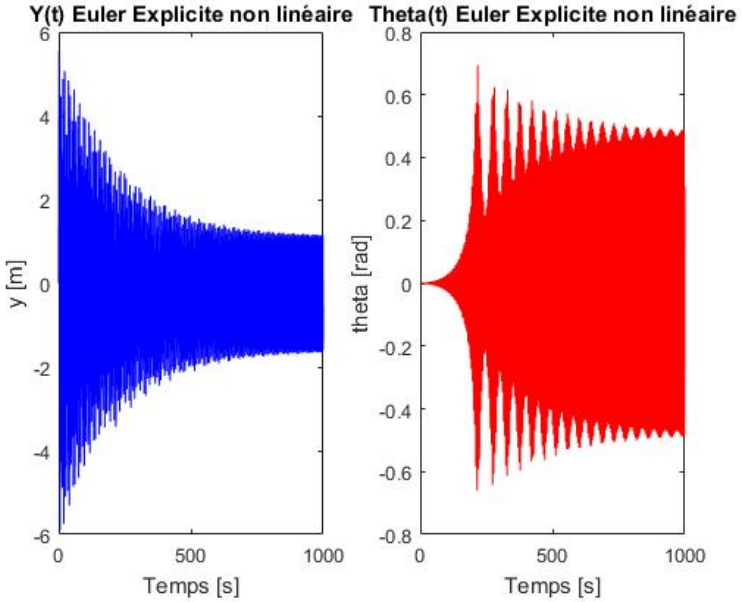
Dans notre code, le pas est désigné par *pas_temps* et *pas_Euler*. La valeur de ce pas de temps influe sur les valeurs des itérés. En effet, nous avons conclu après quelques essais que pour les valeurs de h supérieures à 0.005, les itérés sont très imprécis, voire faux.

2.3.1.Implémentation

L'implémentation de la méthode est semblable à celle de du système d'équations différentielles au point 2.2. Un vecteur T est créé et possède un nombre de lignes égal à la taille du vecteur *t,* passé en paramètre dans les fonctions *explicte_lin* et *explicite_nl*, et quatre colonnes. On affecte les conditions initiales à la première ligne de T, ainsi nous pourrons utiliser ces premières valeurs pour calculer les itérés suivants en utilisant une boucle.

```
for i=2:length(t)
    T(i,:) = T(i-1,:)+ pas_Euler *((odefunction_lin(t(i),T(i-1,:)))');
end
```





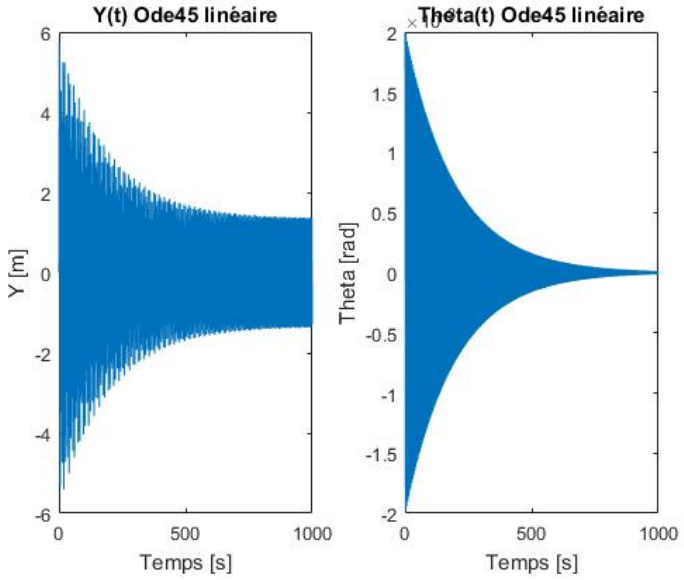
2.4. Fonction Ode45 :

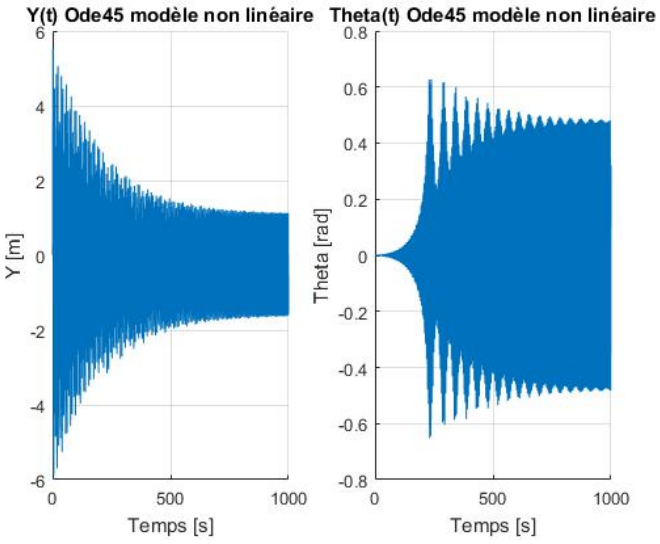
La fonction *Ode45* étant implémentée dans MATLAB, il nous suffisait de choisir une tolérance adaptée et compatible.

```
options = odeset('Reltol', tol);
```

Tout d'abord, nous avons conclu qu'une tolérance relative *Reltol* donnait le même résultat qu'un tolérance absolue *Abstol*. C'est la valeur de la tolérance qui influait sur les différentes valeurs, et donc sur la forme du graphiques. Les valeurs de *tol* supérieures à 0.005 donnent des résultats imprécis et faux.

Nous avons utilisé cette méthode d'essais-erreurs pour déterminer le pas du vecteur *t* optimal.





2.5. Comparaison des méthodes :

La fonction *Ode45* nous donne un résultat plus précis que la méthode d'*Euler explicite*. En effet, le développement de Taylor à l'ordre 1 est bien moins formel que celui de

Runge-Kunta. De plus, un fonction MATLAB par défaut sera toujours meilleur que celles crées par des étudiants amateur.

2.6. Stabilité du pont selon la linéarité et la non-linéarité :

Lorsque les câbles se comportent comme des ressorts (modèle linéaire), l'angle du tablier avec l'horizontale diminue au fil du temps, le pont se stabilise après un péridiode de plus de 1000 secondes. Contrairement au modèle linéaire, où le pont mettra beaucoup plus de temps à se stabiliser.

Question 3 : Étude de la stabilité du tablier de pont :

Dans cette partie, il nous est demandé d'étudier la stabilité du tablier de pont. Pour cela nous devons créer des fonctions dans lesquelles nous ferons varier les paramètres A et α représentant respectivement l'amplitude de la sollicitation cyclique verticale due au vent (N) et le paramètre du modèle du câble (m^{-1}), dans le cas du modèle de câble non linéaire.

3.1 Force exercée dans les câbles :

Pour commencer, nous créons une fonction de prototype '*max_force'* qui prend en argument :

- l'intervalle de temps sur lequel la fonction travaillera, qui sera, ici et ainsi que sur l'ensemble de la question 3, fixé de [0,1000] secondes
- les conditions initiales utilisées dans la question 2, à savoir les valeurs initiales de [y;y';θ;θ'], fixées à [0;0;0,002;0]
- les paramètres A et α.

et qui donne, à sa sortie, la valeur maximale de la force exercée dans les câbles sur la période de simulation considérée.

Cette force est calculée de cette manière : $f(x) = \frac{K}{\alpha}(e^{\alpha x} - 1)$où K=1000 N/m.

Dans un premier temps les valeurs de y et de θ au cours du temps sont calculées à l'aide de la fonction *'ode45_nl'*, créée à la question 2. Cette fonction a été préférée à *'explicite_nl'* pour sa vitesse d'exécution et les meilleures approximations des résultats. Ces valeurs sont utilisées pour calculer, à tous les temps, les valeurs de *f(y+)* et de *f(y-)* où (y+) = y + l*sin(θ) et (y-) = y - l*sin(θ).

Après avoir isolé dans deux vecteurs les valeurs de y et de θ obtenues grâce à la fonction *'ode45_nl'*, les fonctions '*f_yplus'* et *'f_ymin'* sont utilisées pour rendre deux autres vecteurs « fp » et « fm » contenant respectivement les valeurs de f(y+) et de f(y-) tel que $fp_i = y_i + l * \sin(\theta_i)$ et $fm_i = y_i - l * \sin(\theta_i)$ (pour i=1,...,n).
La fonction de MatLab « max » est alors utilisée pour trouver le maximum des valeurs contenues dans « fp » et « fm ».

Le résultat sera la valeur de sortie de la fonction *'max_force'* et, pour des A et α fixés à respectivement 27500 N et $0{,}1m^{-1}$, la fonction renvoie une force maximale dans les câbles de 7842,3 N.

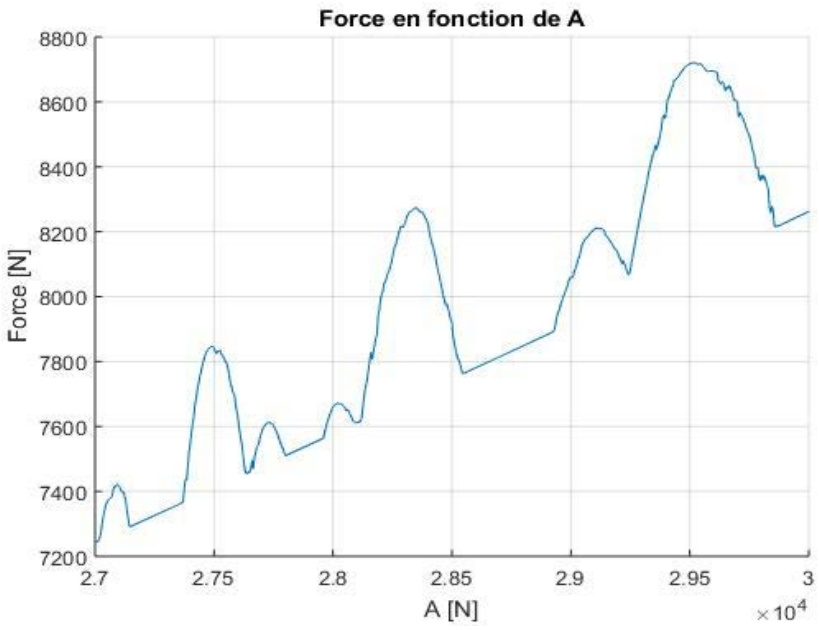


Figure 3.1 : Évolution de la force maximale exercée dans les câbles en fonction de A.

3.2 Calcul de l'amplitude de rupture :

Ensuite, nous créons une fonction qui recherche la valeur de l'amplitude de rupture de A qui, si elle donne lieu à une force de 7900 N dans l'un des deux câbles, provoque leur rupture.

Cette fonction, appelée *'force_foncA'* prend en argument :

- l'intervalle de temps t
- l'intervalle de valeurs de A
- les conditions initiales
- le paramètre α

et applique la méthode de la sécante à la fonction *'max_force'* avec des valeurs x0 et x1 correspondant aux bornes de l'intervalle des valeurs de A, une tolérance de 0.1, un nombre d'itérations maximal de 50 et en appliquant à la fonction les paramètres suivants :

- t = [0;3000] secondes
- A variant de [27000;30000] N , intervalle choisi car, comme le montre le graphique de la figure 3.1, la force maximale dans les câbles n'atteint les 7900 N que sur cet intervalle
- les conditions initiales [0;0;0.002;0]
- α =0.1 m^{-1}

on obtient une amplitude de rupture de 28931 N.

Pour arriver à ce résultat, la méthode de la sécante et a été préférée à celle de la bissection en raison de sa vitesse de convergence plus élevée.

3.3 Evolution de l'amplitude de rupture :

Enfin, nous analysons l'évolution de l'amplitude de rupture A en fonction du paramètre α.

Pour ce faire, dans la fonction appelée *'Question 3'*, nous utilisons une boucle qui fait varier α entre 0.1 m^{-1}et 0.2 m^{-1}avec un pas de 0.005, permettant d'avoir un nombre de données suffisant et un temps de calcul relativement court, et qui pour chaque valeur d' α utilise la fonction *'force_foncA'* pour donner l'amplitude de rupture A sur un intervalle de 27000 N à 30000N, sur une période de 0 à 1000 secondes et avec les mêmes conditions initiales que celles utilisées précédemment.
La fonction affiche ensuite le graphique illustré à la figure 3.2, où l'on remarque que l'évolution de l'amplitude de rupture A est globalement une fonction décroissante d' α, donnant cependant lieu à une asymptote verticale dans le voisinage de α = 0,145m^{-1}.

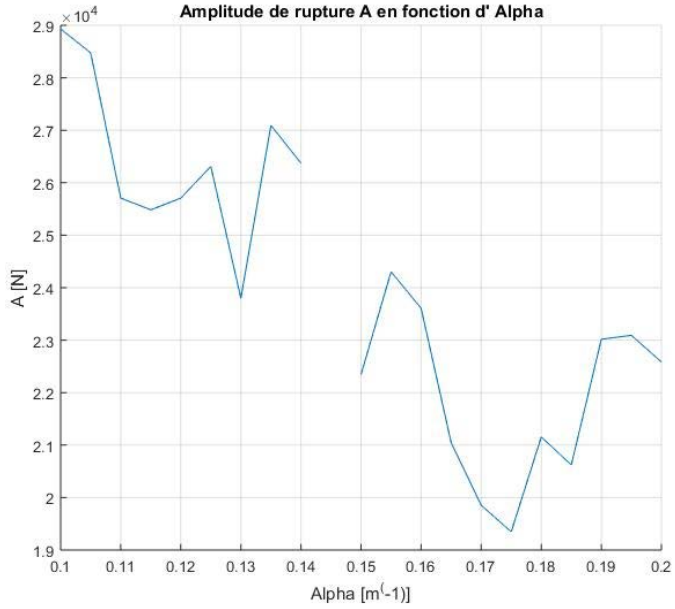


Figure 3.2: Évolution de l'amplitude de rupture A (force maximale dans les câbles de 7900 N) en fonction de α.