CYRIL GARNEAU

SIMPLIFICATION AUTOMATIQUE DE MODÈLE ET ÉTUDE DU RÉGIME PERMANENT

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en génie civil
pour l'obtention du grade de maître ès sciences (M.Sc)

DÉPARTEMENT DE GÉNIE CIVIL
FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2009

Résumé

Les modèles mathématiques servant à simuler le comportement de stations d'épurations représentent un outil puissant pour concevoir une nouvelle installation ou prédire le comportement d'une station d'épuration déjà existante. Cependant, ces modèles ne fournissent aucune information sur un système particulier sans un algorithme pour les solutionner. Il existe actuellement un grand nombre d'algorithmes d'intégration capables de calculer la solution d'un modèle avec précision. Cependant, les temps de calcul en jeux représentent toujours l'un des obstacles à une utilisation extensive des modèles. Deux approches permettent de réduire les temps de calcul, à savoir l'utilisation de matériel informatique plus puissant ou le développement de logiciels et algorithmes plus performants.

L'objectif principal de ce mémoire est de proposer une troisième voie, soit la simplification automatique d'un modèle sur la base de ses valeurs propres. Le Jacobien, une approximation locale du modèle, est utilisé comme base de l'étude des valeurs propres. Une méthode d'homotopie est ensuite utilisée pour maintenir le lien entre les valeurs propres et les variables d'état d'un Jacobien simplifié à sa seule diagonale aux valeurs propres du Jacobien entier. Puisque les valeurs propres représentent une approximation valable de la dynamique des variables d'état d'un modèle, il est possible de trier ces variables d'état sur la base de leurs valeurs propres associées. Les variables d'état présentant une dynamique très rapide par rapport à l'échelle de temps d'intérêt seront alors considérées comme étant toujours à l'équilibre, ce qui permet de négliger leur dynamique transitoire et donc d'accélérer la résolution du modèle. Cette simplification est réalisée à l'intérieur d'un algorithme d'intégration de type Diagonal Implicite de Runge-Kutta capable de résoudre des systèmes d'équation différentielles et algébriques.

Ce mémoire s'attaque également à un cas particulier de la simulation, soit le calcul du régime permanent. Ce calcul peut être réalisé par des algorithmes performants ne recherchant que les valeurs des variables d'état mettant à zéro les équations différentielles. Ces algorithmes sont cependant peu fiables puisque toute solution mathématique est jugée valide, peu importe la réalité physique. La solution proposée est l'injection de connaissance

sous forme de bornes aux valeurs que peuvent prendre les variables d'état. Des équations algébriques implicites sont construites automatiquement sur ces bornes pour forcer la convergence dans l'intervalle voulu.

Abstract

Mathematical models used to simulate the behaviour of wastewater treatment plants are a powerful tool to design a new plant or to predict the behaviour of an existing plant. However, these models do not provide any information without an appropriate numerical solver. Literature provides extensive libraries of algorithms able to solve models with precision. The problem is that computation times are still restraining the use of models. Two approaches allow to reduce computation times, the use of more powerful computers or the development of more efficient software and solvers.

The main objective of this thesis is to propose a third option, the automatic simplification of a model based on the analysis of its eigenvalues. The Jacobian, a local approximation of the model, is used as the basis of the eigenvalue analysis. A homotopy method is then used to maintain the link between the eigenvalues and the state variables from a Jacobian simplified to its main diagonal to the eigenvalues of the full Jacobian. Since the eigenvalues are a valid approximation of the dynamic of the state variables of a model, it is possible to sort the state variables according to their associated eigenvalues. State variables that can be stated fast according to the scale of time of interest are then considered as being at steady state all the time, allowing to neglect transient effects and to fasten the model resolution. The proposed simplification is done inside a Diagonal Implicit Runge-Kutta solver, which is able to manage differential-algebraic equations.

This thesis also works on a specific case of model analysis, the steady-state solution. Powerful solvers exist that are looking for state variable values that nullify the differential equations. Unfortunately, these solvers cannot be considered reliable since any solution is considered valid, bypassing physical reality. The solution proposed to improve the reliability of these solvers consists of injecting knowledge regarding boundaries to the state variables (e.g. concentrations cannot be negative). New implicit algebraic equations are then automatically built with these boundaries to force convergence of the solution to a value within the acceptable range.

Table des Matières

Liste des figures

Liste des tableaux

1 Introduction

Les modèles mathématiques pour prédire le comportement d'usines de traitement biologique d'eaux usées deviennent de plus en plus répandus. Si des modèles standards comme les suites de modèles de boues activées (Activated Sludge Models) ASM (ASM1, ASM2, ASM2d, ASM3, etc) (Henze et al. 2000) ou de digesteurs anaérobiques (Anaerobic Digester Model) ADM (Batstone et al. 2002) sont bien compris par les experts en modélisation, leur solution reste un processus numérique requérant des ressources informatiques importantes.

L'un des objets mathématiques les plus utilisés dans le calcul de la solution d'une simulation numérique reste le Jacobien. Tel qu'il sera vu plus bas, le Jacobien est une matrice des dérivées partielles des fonctions d'état par rapport aux variables d'état. En pratique, cette matrice se veut une approximation du modèle autours du point où elle est calculée. Puisque la manipulation du modèle complet est techniquement fastidieuse, l'analyse de son approximation peut fournir une grande quantité d'information. L'information d'intérêt dans le cadre de cette étude concerne la répartition des différentes dynamiques des variables d'état. Cette analyse est déjà présente dans la littérature. Cependant, elle est habituellement réservée à une analyse à priori du modèle ou encore réservée à des cas particuliers. Aucune méthode générale n'apparaît dans la littérature consultée.

Cette étude utilise donc comme point central l'étude du Jacobien. Les valeurs propres de ce dernier seront analysées dans le but de séparer les équations différentielles originales en deux sections distinctes en fonction de leurs dynamiques propres, soit les variables rapides (associées aux valeurs propres élevées) responsables des longs temps de calculs et les variables lentes (associées aux valeurs propres lentes). Une telle séparation permettra, au niveau de l'intégrateur, de résoudre différemment chaque section. Ainsi, les équations rapides, responsables de la raideur du modèle, seront résolues comme étant à l'équilibre tandis que les variables lentes seront résolues de manière classique. La Figure 1 résume les liens entre les différents éléments de la recherche.

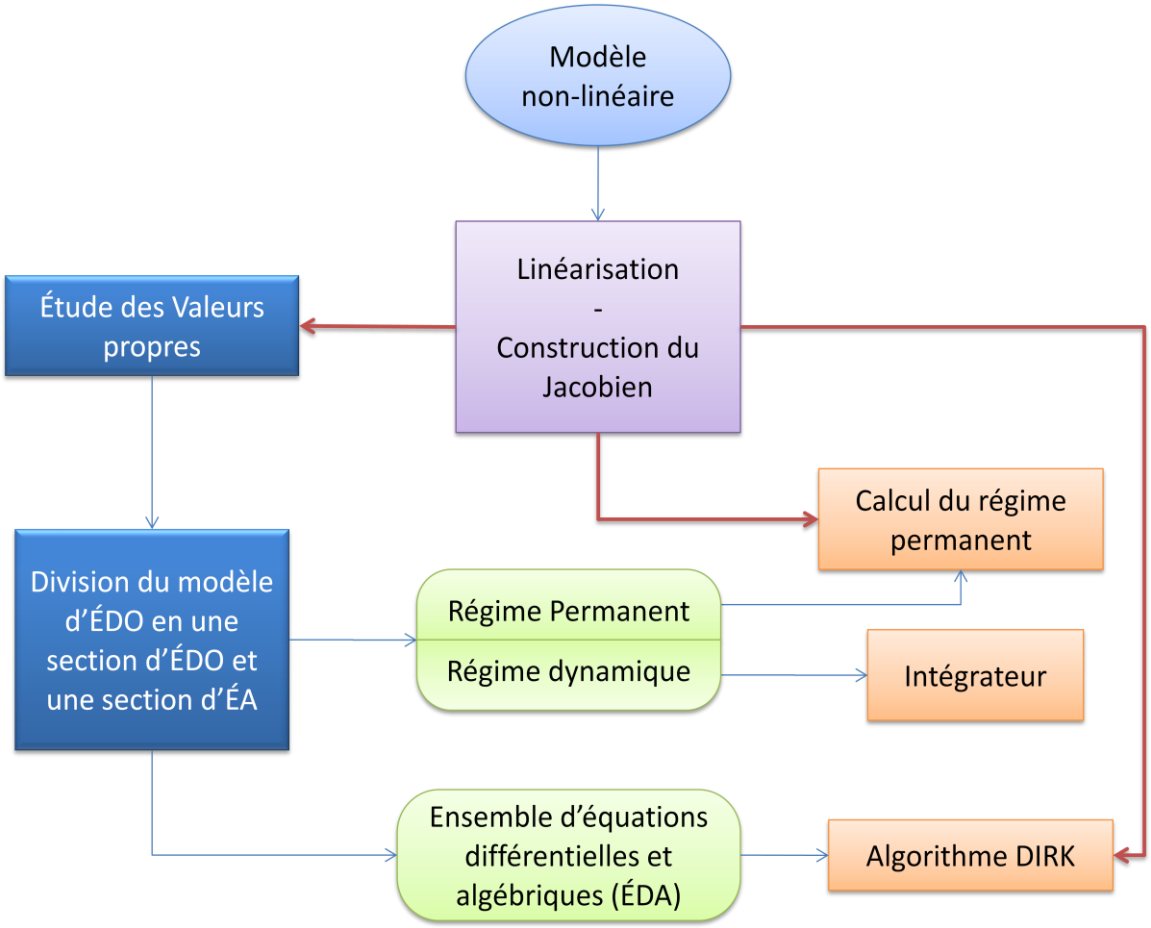


Figure 1 : Lien entre le modèle, le Jacobien, l'analyse du modèle et les différents algorithmes numériques

1.1 Régime dynamique

Le régime dynamique d'un modèle permet d'obtenir les variations temporelles des variables d'état et, ce faisant, d'y inclure des entrées dynamiques. Par exemple, les variations journalières dans la charge polluante à traiter à une station d'épuration.

Malheureusement, si la solution analytique à un problème est toujours la même peu importe les outils mathématiques utilisés pour s'y rendre, il n'en est pas toujours de même avec les techniques numériques. (Seppelt et Richter 2005; Seppelt et Richter 2006) ont démontré qu'avec un simple modèle Prédateur-Proie construit sur les préceptes de Lotka-Volterra, trois solutions dynamiques pouvaient être obtenues selon l'algorithme utilisé, soit : une divergence des variables d'état (leur valeur tendant vers l'infini), une mort de tous les

prédateurs ou encore le schéma habituel cyclique et déphasé entre les populations de proies et de prédateurs. De plus, toujours selon le choix de l'algorithme d'intégration et le choix de ses paramètres initiaux, la durée du cycle pouvait varier du simple au triple. Dans un monde idéal, la solution ne devrait évidement pas dépendre de l'algorithme, mais bien du modèle et des paramètres utilisés.

Un tel comportement illustre la nécessité de bien choisir un algorithme et de bien le régler. Pour le programmeur qui développe l'algorithme, le message encore plus important est de s'assurer de fournir suffisamment d'information sur les paramètres de l'algorithme pouvant être modifiés. En effet, l'étude de (Seppelt et Richter 2005) montre bien que les paramètres par défaut ne seront pas suffisant pour un modèle complexe, d'où la nécessité d'une documentation claire et abondante.

De plus, (Cameron et Gani 1988) et (Claeys 2008b) rappellent que les utilisateurs d'algorithmes d'intégration ne doivent pas être des experts en méthodes numériques pour lancer des simulations dynamiques. Les paramètres modifiables des intégrateurs devraient donc être gardés à leur plus simple expression ou une aide complète sur les différents paramètres de l'intégrateur doit être offerte à l'utilisateur (Claeys 2008b).

1.2 Régime permanent

Le régime permanent est un cas particulier du régime dynamique où les entrées sont constantes et où les variables d'état ne présentent aucune variation dans le temps. L'avantage du régime permanent vient du temps de calcul minimal nécessaire pour obtenir une solution comparativement aux solutions en régime dynamique. Malheureusement, en pratique, il existe peu d'algorithmes permettant de calculer ce régime permanent de façon sure pour les modèles typiquement utilisés pour la modélisation de stations d'épurations. En effet, la taille des modèles couplée à la complexité et à la non-linéarité de ceux-ci génère souvent des réponses multiples mais correctes du point de vue mathématique dont seulement quelques unes possèdent un sens physique. Cette profusion potentielle de réponse est en partie causée par le fait que les contraintes sur les variables d'état ne sont pas appliquées lors de la recherche de la solution. L'exemple le plus fréquent étant de retrouver des concentrations négatives (ce qui ne signifie qu'un échec de l'algorithme).

L'usage le plus courant du régime permanent d'un modèle est pour la conception et le dimensionnement d'une future station d'épuration. L'ingénieur peut alors modifier la configuration de base de la station jusqu'à ce que les résultats lui semblent satisfaisants.

Cependant, déterminer le régime permanent efficacement est également nécessaire dans plusieurs protocoles pour garantir une reproductibilité des résultats obtenus en régime transitoire. Ainsi, (Copp 2002), dans le modèle BSM (pour Benchmark Simulation Model) exige que les valeurs initiales des variables d'état soient les valeurs du régime permanent lorsque les entrées sont constantes. Un autre cas important dans lequel le régime permanent intervient est dans des modèles mathématiques possédant des équations variant à des vitesses très différentes. De tels modèles sont dit raides (ou « stiff » en anglais). Puisque la vitesse de résolution de la grande majorité des intégrateurs (algorithmes capable de résoudre une simulation en régime dynamique) dépend de la raideur du modèle, une technique de réduction de modèle comme l'Approximation du modèle Quasi-Stationnaire (Hesstvedt et al. 1978) (cette section sera abordée en détails plus loin dans le texte) propose de décomposer ce modèle en trois sections.

Ces sections contiendront traditionnellement les équations :

- Rapides, qui seront considérées comme atteignant leur valeur d'équilibre très rapidement. Elles seront résolues par un algorithme de régime permanent.
- Dont l'effet transitoire est requis. Elles seront résolues par un intégrateur classique.
- Variant lentement : Elles seront considérées comme constantes.

C'est pourquoi des techniques fiables et stables doivent être recherchées pour le calcul du régime permanent. Parmi les pistes à suivre, la plus prometteuse est d'intégrer au maximum des connaissances externes au modèle pour favoriser sa convergence vers les solutions acceptables.

1.3 Besoins de simplifier les modèles

L'augmentation de la puissance des ordinateurs sur les quarante dernières années ne peut être mise en doute. Parallèlement, les modèles numériques se sont complexifiés dans le but de profiter pleinement des possibilités des ordinateurs (Gujer 2006). Les simulations

numériques peuvent présentement se classer en deux grandes catégories, les rapides et les longues.

Celles pouvant être réalisées en moins de deux minutes appartiennent à la première catégorie. Leur avantage vient du fait qu'elles peuvent être utilisées pour travailler en temps réel (Alex 2008). Dans ce cas, des calibrations sommaires de modèle peuvent être réalisées sans l'aide d'algorithmes complexes puisque l'effet des paramètres est immédiatement visible.

Les simulations (ou toute expérimentation virtuelle) longues sont généralement planifiées à l'avance et prendront plus de temps à être calculées (de plusieurs dizaines de minutes à quelques jours).

Dans tous les cas, les modélisateurs ont généralement un horizon de temps de calcul jugé normal et construisent leurs modèles en conséquence. Ainsi, une personne souhaitant travailler en temps réel prendra le modèle le plus complexe que son ordinateur est capable de résoudre en moins de deux minutes. Dans ces conditions, la simplification de modèle restera toujours une avenue intéressante puisqu'un modèle « complexe » simplifié pourra donner plus d'information qu'un modèle initialement choisi pour sa simplicité, quitte à confirmer certains résultats à l'aide de simulations utilisant le modèle complet.

L'une des simplifications les plus étudiées est l'analyse en régime quasi-stationnaire. Il s'agit simplement de considérer les variables d'état variant très vite comme étant au régime permanent. Dans ces conditions, un temps de calcul considérable peut être épargné. Cependant, (Farrow et Edelson 1974) mettent en garde contre une utilisation systématique de cette analyse. Ils ont observé des divergences entre la solution témoin et une solution générée grâce à cette analyse qui sont inacceptables. Néanmoins, (Hesstvedt et al. 1978) estiment qu'un contrôle serré de l'analyse en régime quasi-stationnaire permet toujours d'obtenir d'excellentes performances tant en précision des résultats qu'en temps de calcul. Dans l'espoir d'accélérer les temps de calculs qui sont toujours critiques, cette avenue sera explorée pour refléter les problématiques particulières des modèles de traitement biologique des eaux.

1.4 Objectifs

Ce mémoire étudiera de nombreux aspects des méthodes numériques en simulation. Tout d'abord, le calcul d'une simulation dynamique requiert énormément de ressources. L'une des principales difficultés en simulation étant la gestion de la raideur d'un modèle, une technique de simplification automatique sera développée. Cette simplification repose sur le concept d'échelle de temps d'intérêt du modélisateur. En effet, une simulation de plusieurs jours d'une station d'épuration n'a pas besoin d'être précise à la seconde près. Ainsi, il doit être possible de considérer à l'équilibre les variables ayant des constantes de temps très rapides pour ne se concentrer que sur les variables ayant des constantes de temps de l'ordre de l'échelle de temps d'intérêt. L'outil de base de l'évaluation des constantes de temps sera l'analyse des valeurs propres du Jacobien du modèle.

La simplification automatique du modèle mène à un changement fondamental dans les caractéristiques du modèle (voir section 2.1.1.8). Le modèle passe effectivement d'un système d'équations différentielles ordinaires (ÉDO) à un système d'équations différentielles et algébriques (ÉDA). Puisqu'aucun algorithme capable de solutionner des ÉDA n'est disponible sur la plateforme Tornado, un tel algorithme devra être construit à partir de la littérature. Pour ses propriétés numériques, l'algorithme Diagonal Implicite de Runge-Kutta (DIRK) a été choisi (Cameron 1983). De plus, cet algorithme nécessite la solution d'équations implicites non-linéaires. Le Jacobien est donc très présent dans son implémentation puisqu'il est à la base de nombreuses stratégies de solution d'équations implicites non-linéaires.

Un cas particulier de la simulation dynamique sera ensuite étudié, soit le régime permanent. Le régime permanent est atteint lorsque toutes les variables d'état sont à l'équilibre. Compte tenu de son importance dans plusieurs situations, des algorithmes spécifiques ont été développés pour le calculer efficacement. Malheureusement, dans le cadre des modèles de station d'épuration, ces algorithmes offrent des performances généralement inacceptables. Des solutions seront proposées pour améliorer leur performance. Il est à noter qu'une fois encore, le Jacobien est l'instrument de choix de ces algorithmes. Une

bonne compréhension de cet outil permettra d'améliorer les performances des algorithmes en question.

Finalement, compte-tenu de l'importance du Jacobien dans les présents travaux, une tentative est faite de calculer ce dernier symboliquement. Sachant que le Jacobien utilisé dans les algorithmes est généralement une approximation numérique, le fait de connaître celui-ci exactement devrait permettre des gains en précision dans le reste des calculs.

2 Revue de littérature

2.1 Régime dynamique

2.1.1 Algorithmes d'intégration

Les modèles mathématiques de traitement biologique des eaux usées se représentent généralement sous la forme d'équations différentielles ordinaires (ÉDO, ou ODE pour Ordinary Differential Equations en anglais), soit sous la forme :

$$\vec{y}' = \vec{f}(\vec{y}, t) \tag{1}$$

Cependant, certains modèles offrent d'autres formes, que ce soit celle d'équations différentielles algébriques (ÉDA, ou DAE pour Differential-Algebraic Equations en anglais) sous la forme :

$$\begin{aligned} \vec{y}' &= \vec{f}(\vec{y}, \vec{z}, t) \\ 0 &= \vec{g}(\vec{y}, \vec{z}, t) \end{aligned} \tag{2}$$

Ainsi, un modèle sous forme d'ÉDA contient une section d'équations différentielles et une section d'équations algébrique qui nécessite une solution implicite. Tant les modèles d'ÉDO et d'ÉDA nécessitent des conditions initiales pour être résolus.

2.1.1.1 Intégration Numérique

La première technique numérique de résolution d'ÉDO porte le nom d'algorithme d'Euler, du nom du mathématicien Suisse Leonhard Euler (1707-1783). Elle part de la définition d'une équation différentielle :

$$\frac{d\vec{y}}{dt} = \lim_{\Delta t \to 0} \frac{\vec{y}(t + \Delta t) - \vec{y}(t)}{\Delta t} = \vec{f}(t) \tag{3}$$

Si la limite est relaxée et qu'une valeur initiale de y est fournie à $t = t_0$ de même qu'un pas de temps $h = \Delta t$, il est possible de faire avancer la solution à $t = t_0 + h$ à l'aide de l'équation suivante :

$$\vec{y}_1 = \vec{y}_0 + \vec{f}(\vec{y}_0, t_0) * h \quad (4)$$

Il est ainsi possible de calculer une succession de valeurs de $\vec{y}$ entre $t_{initial}$ et t_{final} .

L'algorithme d'Euler, tel que présenté à l'équation (4), est le plus simple. Cependant, si le pas de temps h est trop grand, la solution diverge et ne permet plus de représenter la solution exacte de l'équation différentielle. La construction mathématique de base pour étudier un modèle, tant au niveau de sa stabilité que de la difficulté potentielle à le résoudre est le Jacobien. Une étude approfondie de ce Jacobien est donc essentielle pour étudier un modèle particulier.

2.1.1.2 Étude du Jacobien

Le Jacobien d'un modèle est la matrice des dérivées partielles des fonctions d'état par rapport aux variables d'état :

$$J = \begin{bmatrix} \frac{df_1}{dy_1} & \cdots & \frac{df_1}{dy_n} \\ \vdots & \ddots & \vdots \\ \frac{df_n}{dy_1} & \cdots & \frac{df_n}{dy_n} \end{bmatrix} \quad (5)$$

Si le modèle est linéaire, le Jacobien est constant dans le temps et représente exactement le modèle. Puisque les modèles de stations d'épuration sont non-linéaires par nature, le Jacobien devient une approximation du modèle en un point d'opération $(\vec{y}_i, t_i)$. Ainsi, l'équation (6) peut être approximé autour du point d'opération $\vec{y}_i$ au temps t_i par l'équation (7)

$$\vec{y}' = \vec{f}(\vec{y}, t) \quad (6)$$

$$\vec{y}' = \vec{f}(\vec{y}, t) \approx \vec{f}(\vec{y}_i, t_i) + J_{(\vec{y}_i, t_i)} * (\vec{y} - \vec{y}_i) \quad (7)$$

Puisque les modèles décrivant les stations d'épuration sont généralement non-linéaires, cette approximation est valide pour $\vec{y}$ très proche de $\vec{y}_i$ et pour un temps proche de t_i durant la simulation. Néanmoins, le Jacobien permet de décrire le comportement du modèle autour du point considéré.

Outre la linéarisation du modèle, l'information la plus utile qui peut être extraite du Jacobien demeure ses valeurs propres. Ces dernières donnent en effet de l'information sur la stabilité du modèle ainsi que sur les constantes de temps de ses variables d'état (Steffens et al. 1997).

Les travaux de (Steffens et al. 1997) montrent en effet qu'il est possible de lier chaque valeur propre à une variable d'état. Ce faisant, le modèle n'est plus non seulement linéarisé autour de y_0, mais il est également découplé puisque le Jacobien devient :

$$J \approx \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{bmatrix} \tag{8}$$

Bien sûr, si calculer le modèle à l'aide du Jacobien est une approximation du modèle réel et si ce dernier est réduit à une matrice diagonale, il est possible de douter de la précision du nouveau modèle ainsi construit. Cependant, un tel modèle est si simple qu'il peut être résolu analytiquement. En effet, chaque fonction d'état peut être réécrite comme en (9) :

$$\vec{y}' = \vec{f}(\vec{y}, t) \approx \vec{f}(\vec{y}_0, t_0) + \lambda(\vec{y} - \vec{y}_0) \tag{9}$$

Cette équation peut être simplifiée dans le cadre d'une analyse des valeurs propres en éliminant les constantes. Cette simplification se justifie simplement par le fait que le modèle résultant ne servira qu'à analyser le comportement du modèle original, pas à le résoudre. Ainsi, l'équation (9) peut être simplifiée une dernière fois à :

$$y' = \lambda y \tag{10}$$

Ou encore, à :

$$y' - \lambda y = 0 \tag{11}$$

L'équation 11 est très simple et sa solution exacte est connue depuis longtemps :

$$y = Ae^{\lambda t} \tag{12}$$

La valeur propre associée à une variable d'état donne donc beaucoup d'information sur l'évolution à court terme du modèle. Ainsi, une valeur propre de signe positif signifie que

la variable d'état est instable, i.e. que sa valeur augmentera indéfiniment dans le temps. Une valeur propre négative, à l'opposé, est synonyme de stabilité puisque la variable d'état tendra vers zéro (ou toute autre constante d'intégration non affichée ici). Cette analyse est très bien synthétisée par (Steffens et al. 1997) et représentée à la Figure 2.

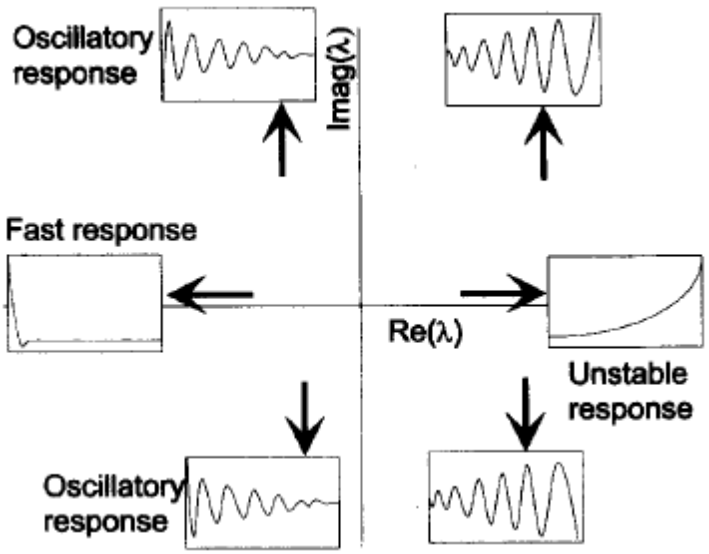


Figure 2 : Réponse transitoire d'une variable d'état en fonction de sa valeur propre (figure tirée de (Steffens et al. 1997))

Si une valeur propre complexe est trouvée, le court développement mathématique qui suit montre que la partie imaginaire est responsable de l'aspect oscillatoire de la réponse :

$$y = Ae^{\left(Re(\lambda)+i*Im(\lambda)\right)*t} \tag{13}$$

$$y = Ae^{Re(\lambda)t}e^{i*Im(\lambda)t} \tag{14}$$

Connaissant la formule d'Euler (15) :

$$e^{i*C} = \cos(C) + i * \sin(C) \tag{15}$$

Il est possible de réécrire (13) une dernière fois :

$$y = Ae^{Re(\lambda)t} * \left(\cos\left(Im(\lambda t)\right) + i * \sin\left(Im(\lambda t)\right)\right) \tag{16}$$

Ainsi, même en ne considérant que la partie réelle de la solution, l'aspect oscillatoire apparaît dès que la valeur propre possède une partie imaginaire

Une dernière information peut être tirée de la valeur propre, soit une approximation de la constante de temps de la variable. La constante de temps d'une équation linéaire est définie comme étant le temps nécessaire pour qu'une variable atteigne 63% de sa variation maximale. Cette valeur est déterminée par l'équation (12) comme étant :

$$\tau = -\frac{1}{\lambda} \tag{17}$$

Autrement dit, une valeur propre négative très élevée est synonyme d'une constante de temps très courte, donc d'une cinétique très rapide. Une valeur propre positive résulte quant à elle en une dynamique instable. L'équation d'état résultante ne possède donc pas de constante de temps.

2.1.1.3 Stabilité des algorithmes d'intégration

La stabilité d'un algorithme d'intégration représente sa capacité à converger vers la solution exacte et à contrôler la propagation de l'erreur sur la solution. La stabilité est habituellement évaluée sur le modèle simple et général de l'unique équation différentielle linéaire (18).

$$y' = \lambda y \tag{18}$$

Le développement mathématique permettant de borner le pas de temps maximal est bien décrit par (Hangos et Cameron 2001). Ainsi la méthode d'Euler est stable tant que la relation (19) est vraie.

$$-2 < h\lambda < 0 \tag{19}$$

En pratique, λ est égal à la plus grande valeur propre négative du Jacobien du modèle. Ainsi, la méthode d'Euler serait contrainte à utiliser de très courts pas de temps si les valeurs propres sont très grandes.

Les algorithmes de Runge-Kutta implicites (abordés plus en détails plus bas dans le texte) sont capables d'atteindre une stabilité telle que (20) est possible.

$$-\infty < h\lambda < 0 \qquad (20)$$

Ainsi, peu importe la longueur du pas de temps utilisé, la solution du modèle sera stable.

Cependant, si le pas est pris trop grand, la précision écopera. La stabilité d'un algorithme satisfaisant (20) est habituellement dit A-stable ou L-stable. La différence est expliquée dans (Bui 1979) comme suit :

Une méthode A-stable verra le rapport entre deux itérations successives tendre vers -1 si le pas de temps tend vers l'infini.

$$\frac{x_{n+1}}{x_n} \to -1 \text{ si } h\lambda \to -\infty \qquad (21)$$

Autrement dit, la solution oscillera entre deux valeurs sans converger, mais sans diverger non-plus.

Une méthode L-stable, quant à elle, verra le rapport entre deux itérations successives tendre vers zéro si le pas de temps tend vers l'infini.

$$\frac{x_{n+1}}{x_n} \to 0 \text{ si } h\lambda \to -\infty \qquad (22)$$

Les propriétés d'une méthode L-stable sont très désirables puisqu'il est clair que la solution n'oscillera pas en augmentant le pas. Malheureusement, une telle méthode est difficile à construire. Dans la littérature, les méthodes A-stables sont de loin les plus courantes et très peu d'implémentations sont proposées pour des méthodes strictement L-stables.

Il existe également d'autres classes de stabilité, mais leur usage est généralement marginal ou utilisé pour décrire une méthode précise. (Hangos et Cameron 2001) en documente quelques unes.

2.1.1.4 Algorithmes d'intégration explicites

Un algorithme d'intégration explicite représente un algorithme qui est capable de déterminer directement la solution de $\vec{y}_{n+1}$. C'est le cas de l'algorithme d'Euler tel qu'écrit à l'équation (4). Cependant, le premier algorithme vraiment efficace pour résoudre des ÉDO fut proposé par MM. Runge et Kutta en 1901. La méthode, qui porte leur nom, considère une somme pondérée des pentes des variables d'état en différents points entre t_n et t_{n+1}. Bien que « Runge-Kutta » réfère généralement à un ensemble de méthodes numériques, par abus de langage, la méthode de Runge-Kutta fait référence à la méthode RK4 suivante :

$$\vec{y}_{n+1} = \vec{y}_n + \frac{1}{6} * h * (k_1 + 2k_2 + 2k_3 + k4) \tag{23}$$

où :

$$\begin{aligned} k_1 &= \vec{f}\,(\vec{y}_n, t_n) \\ k_2 &= \vec{f}\left(\vec{y}_n + \frac{1}{2}hk_1, t_n + \frac{1}{2}h\right) \\ k_3 &= \vec{f}\left(\vec{y}_n + \frac{1}{2}hk_2, t_n + \frac{1}{2}h\right) \\ k_4 &= \vec{f}\,(\vec{y}_n + hk_3, t_n + h) \end{aligned} \tag{24}$$

Cet algorithme encore régulièrement utilisé aujourd'hui est dit d'ordre 4, c'est-à-dire que l'erreur réalisée à chaque pas est d'au plus h^5. Des schémas d'ordre supérieurs ont été développés (comme la méthode de Cash-Karp (Press et al. 1994)), mais la grande popularité de la méthode d'ordre 4 en fait un incontournable des méthodes numériques (Press et al. 1994).

Une analyse de la stabilité de la méthode RK4 montre que sa stabilité est légèrement supérieure à l'algorithme d'Euler et la relation (25) donne les bornes sur le pas de temps en fonction du modèle.

$$-3 \leq h\lambda \leq 0 \tag{25}$$

Ainsi, même si l'algorithme de Runge-Kutta est très performant sur un grand nombre de problèmes, certains modèles sont pratiquement incalculables à cause de temps de calculs prohibitifs dus au manque de stabilité de l'algorithme.

Ces modèles sont dit Raides (ou « stiffs » en anglais) et nécessitent des pas de temps beaucoup trop courts sur un algorithme comme RK4.

2.1.1.5 Algorithmes capable de résoudre des modèles raides

Il existe plusieurs définitions de la raideur. Cependant, la plus courante est basée sur le rapport de la plus grande valeur propre du Jacobien divisé par la plus petite (Cameron et Gani 1988):

$$Raideur = \frac{\lambda_{max}}{\lambda_{min}} \tag{26}$$

Un rapport élevé signifie que certaines variables d'état varient très rapidement comparativement aux plus lentes. Les modèles biologiques entrent généralement dans la catégorie des modèles raides, spécialement lorsque des réactions chimiques comme le calcul du pH sont décrites par des équations différentielles dans le même modèle que des réactions biologiques comme la croissance de la biomasse qui peuvent prendre des jours à se stabiliser (Rosen et al. 2005). Il faut donc exécuter la simulation sur un temps très long pour observer les effets de toutes les variables alors que l'algorithme doit prendre un très court pas de temps pour rester stable.

Les algorithmes capables de résoudre efficacement les systèmes raides sont nombreux et font appel à des techniques très différentes. L'algorithme CVODE, par exemple, utilise une méthode à pas multiples. C'est-à-dire que le pas $\vec{y}_{n+1}$ sera calculé à l'aide des pas $\vec{y}_n$, $\vec{y}_{n-1}$, $\vec{y}_{n-2}$, etc. L'information des pas passés est utilisée pour construire un polynôme d'ordre supérieur à 1 et donc de réduire l'erreur sur le prochain pas. Les méthodes à pas multiples les plus connues sont les algorithmes d'Adams-Bashforth, Adams-Moulton et les formules de différentiation arrière (Backward Differentiation Formulas ou BDF en anglais).

Une seconde méthode utilisée fréquemment pour résoudre les systèmes raides est la méthode d'extrapolation de Richardson bien décrite dans (Press et al. 1994). Dans ce cas-ci,

un pas de temps fixe H est utilisé pour calculer $\vec{y}_{n+1}$. L'erreur est estimée en intégrant ce même pas de temps avec deux pas intermédiaires deux fois plus courts ($h = H/2$). Si le calcul est recommencé avec trois pas de temps, puis quatre, il est possible de voir converger l'erreur vers zéro. La méthode d'extrapolation de Richardson permet donc de déterminer la valeur de $\vec{y}_{n+1}$ en extrapolant le point où l'erreur est nulle (donc calculée avec un nombre de pas intermédiaires qui tend vers l'infini) à partir des erreurs estimées en $E_H, E_{h=\frac{H}{2}}, E_{h=\frac{H}{3}},$ etc. La méthode de Burlisch-Stoer utilise cet algorithme.

Malheureusement, si les méthodes à pas multiple et d'extrapolation peuvent résoudre efficacement des systèmes d'ÉDO raides, leurs performances se dégradent très rapidement si les équations différentielles présentent des discontinuités (interrupteurs, contrôleurs, bruit dans les entrées, etc.) ou des singularités (Press et al. 1994). En effet, ces méthodes construisent un polynôme sur plusieurs points dans le temps. Si la fonction n'est pas suffisamment lisse, un polynôme d'ordre élevé risque de diverger rapidement lorsqu'il est utilisé pour extrapoler le point suivant.

2.1.1.6 Méthode de Runge-Kutta implicite

Les méthodes de Runge-Kutta implicites permettent de contourner les inconvénients des méthodes présentées ci-dessus puisqu'elles n'utilisent pas d'information passée et qu'elles utilisent une somme pondérée pour calculer le prochain pas, évitant les risques inhérents aux polynômes d'ordre élevés. Finalement, le concept de méthode implicite garantie une stabilité numérique à la majorité des systèmes d'ÉDO.

L'une des méthodes de Runge-Kutta implicite les plus populaires est la méthode Diagonale Implicite de Runge-Kutta (DIRK). Elle se définit par la matrice (27) :

$$\begin{array}{c|ccc} c_1 & a_{11} & & \\ \vdots & \vdots & \ddots & \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} \qquad (27)$$

où les coefficients a_{ii} sont identiques et sont désignés par la lettre γ. Les coefficients de (27) sont ensuite insérés dans les équations (28), (29) et (30) :

$$\vec{y}_{n+1} = \vec{y}_n + h\sum_{i=1}^{s} b_i \vec{f}\left(\vec{y}_{n,i}, t_{n,i}\right) \tag{28}$$

$$\vec{y}_{n,i} = \vec{y}_n + h\sum_{j=1}^{i-1} a_{ij}\vec{f}\left(\vec{y}_{n,j}, t_{n,j}\right) + \gamma h\vec{f}\left(\vec{y}_{n,i}, t_{n,i}\right) \tag{29}$$

$$t_{n,i} = t_n + c_i h \tag{30}$$

Ces deux équations représentent la forme de base d'un algorithme Diagonal Implicite de Runge-Kutta. Puisque l'algorithme se définit par ses paramètres a_{ij}, b_j et c_i, quelques ensembles de paramètres se retrouvent dans la littérature. L'un des plus utilisés vient de (Cameron 1983). Par contre, (Bui 1979) offre un autre ensemble de paramètres alors que (Alexander 2003) rassemble les équations permettant de construire de nouveaux jeux de paramètres.

L'avantage des paramètres proposés par (Cameron 1983) vient de la possibilité de construire quatre méthodes d'ordre 1 à 4 en réutilisant les paramètres a_{ij} et c_i comme suit :

$$\begin{array}{c|c} c_1 & \gamma \\ \hline & b \end{array} \tag{31}$$

$$\begin{array}{c|cc} c_1 & \gamma & \\ c_2 & a_{21} & \gamma \\ \hline & d_1 & d_2 \end{array} \tag{32}$$

$$\begin{array}{c|ccc} c_1 & \gamma & & \\ c_2 & a_{21} & \gamma & \\ c_3 & a_{31} & a_{32} & \gamma \\ \hline & e_1 & e_2 & e_3 \end{array} \tag{33}$$

$$\begin{array}{c|cccc} c_1 & & & & \\ c_2 & \gamma & & & \\ c_3 & a_{21} & \gamma & & \\ c_4 & a_{31} & a_{32} & \gamma & \\ & a_{41} & a_{42} & a_{43} & \gamma \\ \hline & f_1 & f_2 & f_3 & f_4 \end{array} \tag{34}$$

Les paramètres proposés sont affichés au Tableau 1 :

Tableau 1 : Paramètres de l'algorithme DIRK proposés par (Cameron 1983)

γ	0.435866521508	d_1	1.158945191501
a_{21}	-0.403494298165	d_2	-0.158945191501
a_{31}	-0.381596758045	e_1	0.661090792671
a_{32}	0.945730236526	e_2	0.131307259462
a_{41}	0.401916934763	e_3	0.207601947867
a_{42}	-0.110263523009		
a_{43}	-0.163386454770	f_1	0.238148535874
		f_2	0.190784762258
b	1.0	f_3	0.155701460900
		f_4	0.415365240968
c_1	γ		
c_2	0.032372223343		
c_3	1.0		
c_4	0.564133478492		

Finalement, l'équation (29) doit être résolue implicitement puisque le terme $\vec{y}_{n,i}$ se retrouve des deux côtés de l'équation, d'où un temps de calcul significativement plus long que dans le cas des méthodes explicites. Une méthode itérative, habituellement la méthode de Newton-Raphson, est donc nécessaire pour calculer la solution de (35).

$$0 = -\vec{y}_{n,i} + \vec{y}_n + h\sum_{j=1}^{i-1} a_{ij}\vec{f}\left(\vec{y}_{n,j}, t_{n,j}\right) + \gamma h\vec{f}\left(\vec{y}_{n,i}, t_{n,i}\right) \qquad (35)$$

Autrement dit, si quatre étages sont nécessaires pour obtenir une solution respectant les tolérances, quatre ensembles de solutions doivent être calculés à l'aide de la méthode de Newton-Raphson.

Il existe des méthodes de Runge-Kutta implicites dont des éléments situés dans la matrice supérieure sont non-nulles. La famille de méthodes la plus connue est celle de Lobatto. Les méthodes complètement implicites remplacent l'équation (29) par :

$$\vec{y}_{n,i} = \vec{y}_n + h\sum_{j=1}^{s} a_{ij}\vec{f}\left(\vec{y}_{n,j}, t_{n,j}\right) \qquad (36)$$

Ainsi, chaque fois que j est supérieur à i, les variables $\vec{y}_{n,i}$ doivent être calculées implicitement.

2.1.1.7 Solution d'un modèle d'équations différentielles et algébriques

L'algorithme DIRK proposé par (Cameron 1983) possède une propriété supplémentaires, soit la possibilité de résoudre un système d'ÉDA. Pour se faire, les équations (28), (29) et (30) sont réécrites comme suit :

$$\vec{y}_{n+1} = \vec{y}_n + h\sum_{i=1}^{s} b_i \vec{f}\left(\vec{y}_{n,i}, \vec{z}_{n,i}, t_{n,i}\right) \tag{37}$$

$$\vec{y}_{n,i} = \vec{y}_n + h\sum_{j=1}^{i-1} a_{ij}\, \vec{f}\left(\vec{y}_{n,j}, \vec{z}_{n,j}, t_{n,j}\right) + \gamma h \vec{f}\left(\vec{y}_{n,i}, \vec{z}_{n,i}, t_{n,i}\right) \tag{38}$$

$$t_{n,i} = t_n + c_i h \tag{39}$$

Les variables $\vec{z}$ représentent les variables algébriques dont l'équation doit être résolue implicitement.

$$0 = \vec{g}(\vec{y}, \vec{z}, t) \tag{40}$$

Dans le cadre de l'algorithme DIRK, ces équations implicites sont résolues à chaque étape intermédiaire (38) dans le cadre de la solution implicite des équations différentielles et une fois que le pas complet est calculé. Cette dernière étape ne solutionne que les équations algébriques en gardant constantes les variables différentielles.

$$0 = g(y_{n+1}, z_{n+1}, t_{n+1}) \tag{41}$$

Généralement, l'équation (41) est solutionnée rapidement puisqu'un bon estimé initial peut être calculé sur la base des étapes intermédiaires. Quelques itérations (2 à 5) sont généralement suffisantes pour calculer la solution.

Il est à noter que (Cameron 1983) évoque un jeu de paramètre de l'algorithme DIRK différent de celui présenté au Tableau 1 qui permettrait d'éviter d'avoir à solutionner l'équation (41). Malheureusement, ces paramètres ne sont pas fournis dans la publication.

2.1.1.8 Réduction d'un modèle dynamique

2.1.1.8.1 Modèle quasi-stationnaire

La réduction d'un modèle dynamique d'ÉDO ou d'ÉDA (Équations Différentielles et Algébriques) en deux ou trois sections est une procédure courante en génie chimique (Cameron 2008). Dès 1978, (Hesstvedt et al. 1978) rapportent que la réduction d'un modèle raide en deux ensembles d'équations est couramment utilisée. Les premiers algorithmes capables de résoudre efficacement des modèles raides apparaissent à la même époque. Malgré l'amélioration significative des temps de calculs dus à ces algorithmes complexes, (Hesstvedt et al. 1978) rapportent que les temps de calculs sont toujours trop long et qu'une réduction du modèle permet d'améliorer davantage le temps de calcul. Ils notent toutefois que des simulations réalisées à l'aide du modèle complet sont toujours nécessaires pour comparer les résultats obtenus par les modèles réduits.

(Hesstvedt et al. 1978) nomment leur technique de réduction de modèle l'Approximation du Modèle Quasi-Stationnaire (Quasi-Steady-State Approximation en anglais). Dans leur article de 1978, ils utilisent un modèle de pollution atmosphérique et se basent sur les constantes de temps de dissociation des constituants pour les classer dans les équations rapides ou moyennes. Ils utilisent également un pas de temps fixe durant sa simulation de 30 secondes. Dans ces conditions, une réaction est considérée rapide si sa constante de temps est 10 fois inférieure au pas de temps. Une réaction est considérée lente si sa constante de temps est 100 fois supérieure au pas de temps :

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}) = \begin{cases} \vec{f}_1(\vec{y}) & \rightarrow \tau < \dfrac{\Delta t}{10} \\ \vec{f}_2(\vec{y}) & \rightarrow \dfrac{\Delta t}{10} < \tau < 100\Delta t \\ \vec{f}_3(\vec{y}) & \rightarrow 100\Delta t < \tau \end{cases} \tag{42}$$

Ainsi, l'ensemble d'équations $\vec{f}_1$ est considéré au régime permanent à tous les pas de temps, l'ensemble $\vec{f}_2$ est résolu par la méthode de Gear et l'ensemble $\vec{f}_3$ est considéré comme variant très peu et est résolu par l'algorithme d'Euler. Dans ces conditions, chaque section est résolue par l'algorithme qui semble le plus approprié. Finalement, les ensembles

d'équation n'ont pas à être statiques puisque les constantes de temps peuvent varier en fonction des autres variables d'état. Ainsi, la dynamique rapide d'une équation pourra ralentir au point que l'équation soit résolue par la méthode de Gear plutôt que par l'algorithme de régime permanent et vice versa. Parmi les problèmes de la technique proposée, il faut noter l'accès aux équations des constantes de temps. Ainsi, le logiciel utilisé pour simuler le modèle doit posséder une structure lui donnant accès à ces constantes de temps. L'utilisation d'un simulateur voyant le modèle comme une boîte noire nécessiterait donc d'importantes modifications pour utiliser l'approximation du modèle quasi-stationnaire.

2.1.1.8.2 Perturbation Singulière

La technique de la perturbation singulière est une autre technique utilisée pour réduire la complexité d'un modèle mathématique. Bien que le résultat se rapproche de celui observé pour un modèle quasi-stationnaire (un ensemble d'équations différentielles divisé en trois sous-groupes d'équations), la technique offre un formalisme mathématique permettant une étude du modèle avant sa résolution.

La technique, décrite par (Dochain et Vanrolleghem 2001), revient à multiplier une équation différentielle par un paramètre ayant une très faible valeur. Ainsi, la forme originale de l'équation est présentée en (43) :

$$\frac{dy_1}{dt} = f(y_1, y_2, \dots, t) \tag{43}$$

La perturbation singulière est réalisée par l'opération suivante :

$$y_1 \rightarrow \theta * y_{1_{const}} \tag{44}$$

Ce qui transforme l'équation (43) comme suit :

$$\theta * \frac{dy_1}{dt} = f_1(\theta * y_{1_{const}}, y_2, \dots, t) \tag{45}$$

Lorsque la valeur du paramètre est réduite à zéro, l'équation (45) devient une équation algébrique implicite :

$$0 = f_1(y_2, \dots t) \tag{46}$$

La perturbation singulière s'applique à deux cas de figures. Dans un premier cas, la variable à résoudre de façon implicite varie très lentement. Le paramètre θ représente alors la cinétique de la variable. Il est donc acceptable de poser sa cinétique à zéro pour réduire le nombre d'équations différentielles à résoudre simultanément.

$$\theta = k_{min} \approx 0 \tag{47}$$

Dans le deuxième cas, les différentes variables d'état montrent des cinétiques d'ordres de grandeurs très différents. Il est possible, après quelques manipulations algébriques, de définir le paramètre θ comme suit :

$$\theta = \frac{1}{k_{max}} \approx 0 \tag{48}$$

La technique montrée par (Dochain et Vanrolleghem 2001) ne permet pas de détecter les variables d'état à résoudre implicitement. Ces variables doivent donc être identifiées à priori. Par contre, la gestion des variables à extraire est simplifiée puisque les équations modifiées peuvent être traitées algébriquement pour réduire la complexité globale du modèle. Par exemple, les équations implicites peuvent dans certains cas être exprimées sous une forme explicite.

2.1.1.8.3 Concept d'Homotopie

Si les valeurs propres du Jacobien permettent de déterminer les constantes de temps des variables d'état en un point d'opération, elles offrent tout de même une approximation valable de celle-ci sur une plage raisonnable d'utilisation. Il est donc possible de classer les variables d'état en fonction de leur cinétique. Cependant, lorsque les valeurs propres d'une matrice sont calculées, elles sont retournées sous la forme d'un vecteur trié. Il est donc impossible d'associer une valeur propre à une variable d'état directement. (Steffens et al. 1997) proposent d'utiliser une technique d'homotopie pour les lier.

L'homotopie est une déformation continue entre deux objets. Les deux objets que (Steffens et al. 1997) proposent d'utiliser sont le Jacobien du modèle et une matrice de mêmes

dimensions dont seuls les éléments de la diagonale sont identiques à la diagonale du Jacobien, tous les autres éléments étant nuls.

$$H_1 = Jac \tag{49}$$

$$H_2 = \begin{bmatrix} Jac_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & Jac_{nn} \end{bmatrix} \tag{50}$$

Les valeurs propres de la matrice (50) sont les éléments de la diagonale. Le lien entre les différentes variables d'état et leur valeur propre associée est donc trivial. Par la suite, une contribution toujours plus importante du Jacobien total est ajoutée selon l'équation (51) :

$$H = r * H_1 + (1 - r) * H_2 \tag{51}$$

Le paramètre r est le paramètre d'Homotopie. Il varie de zéro (H ne dépend que de la diagonale) à un (H est le Jacobien entier). Les valeurs propres de H sont calculées en variant r plus ou moins rapidement pour permettre de suivre l'évolution du lien entre les valeurs propres de la matrice d'homotopie et les variables d'état.

2.1.1.8.4 Analyse des valeurs propres

(Steffens et al. 1997) proposent d'effectuer l'analyse des valeurs propres avant de commencer à utiliser le modèle. De plus, comme le Jacobien doit être évalué en un point bien précis, les variables d'état sont initialisées au régime permanent avant de l'évaluer. Le régime permanent propose en effet les meilleures valeurs par défaut où évaluer le Jacobien pour une procédure systématique. Cependant, en régime dynamique, un modèle peut être en perpétuel déséquilibre, par exemple dans le cas d'entrées dynamiques simulant l'affluent d'une station d'épuration. Dans ces conditions, l'analyse au régime permanent peut être très éloignée de la réalité au point d'opération. C'est pourquoi (Steffens et al. 1997) proposent d'évaluer le comportement du modèle réduit lorsque des échelons sont imposés en entrée. Cette technique permet de détecter les variables d'état dont la constante de temps s'allonge de façon inacceptable et de les reclasser dans la catégorie de variables à cinétique moyenne.

Bref, la technique proposée par (Steffens et al. 1997) est une méthode numérique d'évaluation du modèle qui doit être réalisée avant toute résolution du modèle. Elle se base

sur l'étude des valeurs propres du Jacobien, ce qui permet un découplage des équations d'état selon leurs constantes de temps, comme à l'équation (42). Bien que plusieurs approximations soient nécessaires (Jacobien évalué lorsque les variables d'état sont au régime permanent, linéarisation du modèle, découplage du modèle linéarisé et retrait des constantes dans la recherche d'une solution analytique) pour porter un jugement, il a été démontré par (Steffens et al. 1997) que les temps de calculs pouvaient être réduits de moitié sans induire d'erreur de plus de 5% par rapport au modèle original. Puisque le modèle utilisé était un modèle ASM1 couplé à un décanteur secondaire divisé en 20 couches, la complexité impliquée est très représentative des modèles plus complexes du domaine de la modélisation du traitement des eaux usées.

2.1.1.8.5 Étude alternative du Jacobien

(Okino et Mavrovouniotis 1999), pour leur part, étudient le Jacobien en posant l'approximation suivante : Puisque les modèles cinétiques de réactions chimiques présentent généralement des matrices Jacobiennes triangulaires inférieures, ou s'en approchent, leurs valeurs propres sont semblables à l'élément sur leur diagonale. Ainsi, il est possible d'obtenir un estimé de la cinétique des variables d'état peu coûteux puisque le calcul du Jacobien intervient dans plusieurs algorithmes d'intégration. Cependant, cette approximation est jugée valide sur des modèles de réactions chimiques. Elle n'est pas démontrée dans le cadre de modèles biologiques ou encore de contrôleurs, qui sont fréquents dans des modèles simulant des usines entières. Il est donc difficile d'extrapoler l'utilité de ces résultats aux modèles de station d'épuration.

2.2 Régime Permanent

Il existe plusieurs stratégies pour la recherche du régime permanent d'un modèle donné. Les deux plus courantes sont sans doute l'exécution d'une simulation dynamique à entrées constantes jusqu'à l'équilibre. Ainsi, le régime permanent est facilement identifiable sur la Figure 3. Après un état transitoire d'environ 6 jours, la variable d'état ne présente plus de variation. Cependant, cette variable d'état répond à l'équation d'état (52) et a pour condition initiale $y_{t=0} = 1$.

$$\frac{dy}{dt} = -y \tag{52}$$

Puisque le régime permanent se définit comme par l'équation (53), il est plus intéressant de rechercher la valeur de y pour laquelle l'équation (52) est nulle. C'est la stratégie des algorithmes de recherche du régime permanent.

$$\frac{d\vec{y}}{dt} = 0 \tag{53}$$

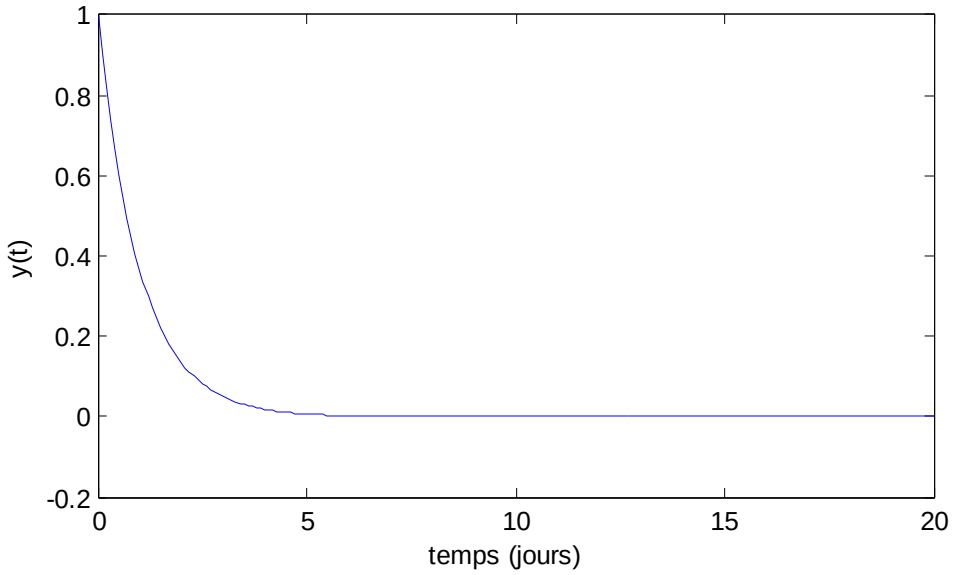


Figure 3 : Variable d'état atteignant l'équilibre dans le temps.

Le choix d'un algorithme dépendra donc de l'information recherchée sur un modèle et de la structure du modèle. Ce choix influencera la précision de cette solution ou encore la vitesse de calcul de la solution. Ce dernier critère devient le facteur principal et le principal problème puisque chaque algorithme doit être configuré selon des critères qui lui sont propre, ce qui demande une excellente connaissance des intégrateurs de la part de l'utilisateur.

Les algorithmes calculant le régime permanent recherchent les valeurs que doivent avoir les variables d'état pour que les fonctions d'état soient nulles. Dans de nombreuses simulations, des variables d'entrées sont nécessaires pour observer un régime dynamique. Bien entendu, pour que le calcul du régime permanent ait un sens, ces entrées doivent être constantes. D'un point de vue mathématique, ces algorithmes résolvent donc un système d'équations comme en (54). Finalement, compte tenu de la définition du régime permanent, le temps n'est pas une variable. Lorsque nécessaire, il est imposé à zéro.

$$\vec{y}' = \vec{f}(\vec{y})_{t=0} = 0 \qquad (54)$$

2.2.1 Utilisation de la simulation dynamique jusqu'au Régime Permanent

Dans plusieurs cas, la méthode la plus simple pour calculer le régime permanent consiste à lancer une simulation dynamique avec des entrées constantes sur une période de temps suffisamment longue pour que les variables d'état atteignent le régime permanent. Cette technique est d'ailleurs recommandée par (Copp 2002) si aucun algorithme de Régime Permanent efficace n'est disponible.

Il s'agit bien sûr de la méthode la plus coûteuse en calcul qui puisse être imaginée, bien que certains algorithmes d'intégration soient très performants et permettent d'obtenir une solution assez rapidement. Il s'agit souvent également de la méthode la plus sure puisque tous les algorithmes d'intégration peuvent remplir cette tâche. De plus, l'utilisation d'un algorithme implicite (ce qui permet l'utilisation de pas de temps large) couplé à des tolérances faibles permet l'atteinte du régime permanent en des temps de calcul raisonnables (Alex 2008). La tolérance faible aura habituellement pour effet de rendre la réponse transitoire imprécise, ce qui n'est pas un problème dans ce cas particulier.

2.2.2 Réduction du modèle en Régime Permanent

Dans plusieurs cas, les modèles à résoudre sont très complexes. Cette complexité se définit soit par une très forte non-linéarité soit par un nombre très important d'équations à résoudre simultanément.

Si la non-linéarité est un problème abordé et habituellement bien géré par la technique de la zone de confiance (« trust-region », voir la méthode Hybride de la bibliothèque MINPACK ci-dessous), le nombre d'équations à résoudre simultanément peut facilement exploser. Dans le domaine des modèles de station d'épuration, l'un des modèles les plus simples, le « Benchmark Simulation Model 1 » (BSM1) (Copp 2002) contient 145 variables d'état. Le simple calcul du Jacobien implique donc de déterminer la valeur de 21025 dérivées partielles. Puisque des modèles plus réalistes d'usines peuvent facilement compter 500 variables d'état, le calcul du Jacobien et sa décomposition LU ou QR peuvent devenir très important (le modèle VLWWTP, ou « Very Large Wastewater Treatment Plant », développé par (Claeys 2008a) en contient 579 et le modèle IUWS, ou « Integrated Urban Wastewater System » en contient, pour sa part, 1663 (Claeys 2008a)). Ainsi, la décomposition LU d'une matrice Jacobienne de 500 variables d'état prend environ $N^2 + \frac{1}{3}N^3$ opérations, soit plus de 40 millions d'opérations.

Dans ces conditions, il est évident que des recherches ont été faites dans le but de diminuer la taille des modèles à résoudre. Ainsi, (Kakizaki et Sugawara 1985) ont proposé de réduire le nombre de variables d'état sur la base de l'étude du Jacobien. Leur étude porte sur l'analyse du Jacobien colonne par colonne en observant qu'une colonne est décrite comme suit :

$$J_{colonne} = \begin{bmatrix} \frac{df_1}{dy_1} \\ \frac{df_2}{dy_1} \\ \frac{df_3}{dy_1} \end{bmatrix} \qquad (55)$$

Lorsque tous les éléments d'une colonne sont nuls ou jugés suffisamment près de zéro, la variable d'état responsable de ces dérivées (y_1 dans l'équation (55)) est extraite puisque la solution des autres variables d'état ne dépend pas, ou très peu, de la variable extraite.

Un tel système est en fait surdéterminé, c'est-à-dire qu'il existe plus d'équations que d'inconnus. Dans ces conditions, le régime permanent de la fonction d'état peut ne pas exister. S'il existe, il sera calculé à partir des autres variables d'état à l'équilibre comme

dans l'équation suivante, en considérant que la variable y_1 ne donne que des dérivés partielles nulles :

$$y_1^{'} = f_1(y_2, y_3, y_4, \dots) \tag{56}$$

L'étude a été réalisée dans le domaine des circuits électroniques. Cependant, les auteurs affirment qu'une majorité des variables d'état extraites étaient des éléments parasites ou ayant une influence négligeable sur leurs circuits. L'effet sur les résultats globaux peut donc varier, mais la convergence est observée plus fréquemment sur les problèmes réduits.

2.2.3 Algorithmes de résolution de système non-linéaire

Le régime permanent d'un modèle est calculé en posant toutes les équations différentielles de ce modèle à zéro

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}) = 0 \tag{57}$$

Ainsi, au régime permanent, la variation des variables d'état y dans le temps est nulle. La valeur de y_{RP} est dite la racine de l'équation $\vec{f}(\vec{y})$.

Si, dans des cas simples, une solution analytique peut être trouvée, lorsque $\vec{f}(\vec{y})$ est non-linéaire, il faut souvent recourir à une méthode numérique pour obtenir une approximation de la racine de $\vec{f}(\vec{y})$.

2.2.3.1 Méthodes en une dimension

2.2.3.1.1 Méthode de la Bissectrice

Plusieurs méthodes ont été développées en une dimension (ou une variable). La plus simple étant probablement la méthode de la Bissectrice :

1. Trouver deux points $y_1$et $y_2$tels que $y_1 \times y_2 < 0$, i.e. une valeur de y est positive et l'autre est négative.
2. Calculer $f(y_3) = f\left(\frac{y_1+y_2}{2}\right)$
3. Si $y_1 \times y_3 > 0$

$$y_1 = y_3$$

4. Sinon

$$y_2 = y_3$$

5. Tant que

$$y_2 - y_1 > tol$$

retourner en 1

Ainsi, dès qu'une racine est encadrée, l'algorithme de la bissectrice converge vers cette racine. Bien qu'il existe des cas pathologiques comme une fonction rationnelle où la prétendue racine est en fait une asymptote, la méthode de la bissectrice demeure l'une des plus robustes.

2.2.3.1.2 Méthode de Newton-Raphson

La méthode de la bissectrice est généralement plus académique que pratique. En pratique, lorsque la racine d'une fonction est recherchée, c'est la méthode de Newton-Raphson qui est la plus utilisée. Cette dernière utilise un point de départ y_0, la fonction évaluée en ce point $f(y_0)$ ainsi que la pente de la fonction en ce point $\frac{df(y_0)}{dy}$. L'algorithme se définit comme suit :

1. Calculer $y_0, f(y_0)$ et $\frac{df(y_0)}{dy}$
2. Tant que $y_{i+1} - y_i > tol$

$$y_{i+1} = y_i - \frac{f(y_i)}{\left(\frac{df(y_i)}{dY}\right)}$$

L'ordre de convergence de la méthode de Newton-Raphson est dit quadratique puisqu'il correspond à la définition (58) :

$$\frac{\|y_{k+1} - y^*\|}{\|y_k - y^*\|^2} \leq c \qquad c \geq 0, k \gg 1 \qquad (58)$$

où k est la k^e itération de l'algorithme et où y* est la racine exacte et c est une constante positive quelconque.

Bien sûr, l'algorithme de Newton-Raphson n'est pas sans risque. En effet, si la dérivée en un point calculé est nulle, l'algorithme est incapable de converger vers une racine. De plus, pour être efficace, l'estimé initial y_0 doit être « suffisamment » près de la racine et la fonction doit être « suffisamment » bien conditionnée et ne pas présenter de comportement pathologique. Bien que ces cas soient décrit dans la plupart des ouvrages d'analyse numérique, (Press et al. 1994) en couvre les grandes lignes. Pour une fonction sinusoïdale, par exemple, si l'estimé initial est suffisamment proche de la racine, l'algorithme effectuera le tracé de la Figure 4.

Si l'estimé initial n'est pas dans la zone de convergence de la racine, un comportement différent pourra être observé, comme à la Figure 5:

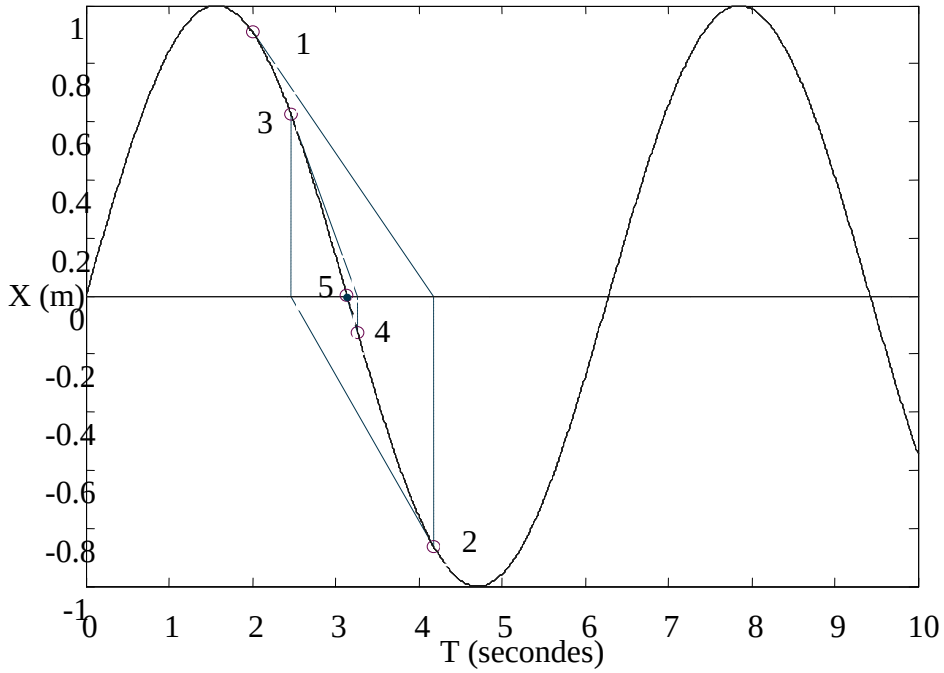


Figure 4 : Convergence de la méthode de Newton-Raphson à la racine la plus proche sur une fonction sinusoïdale

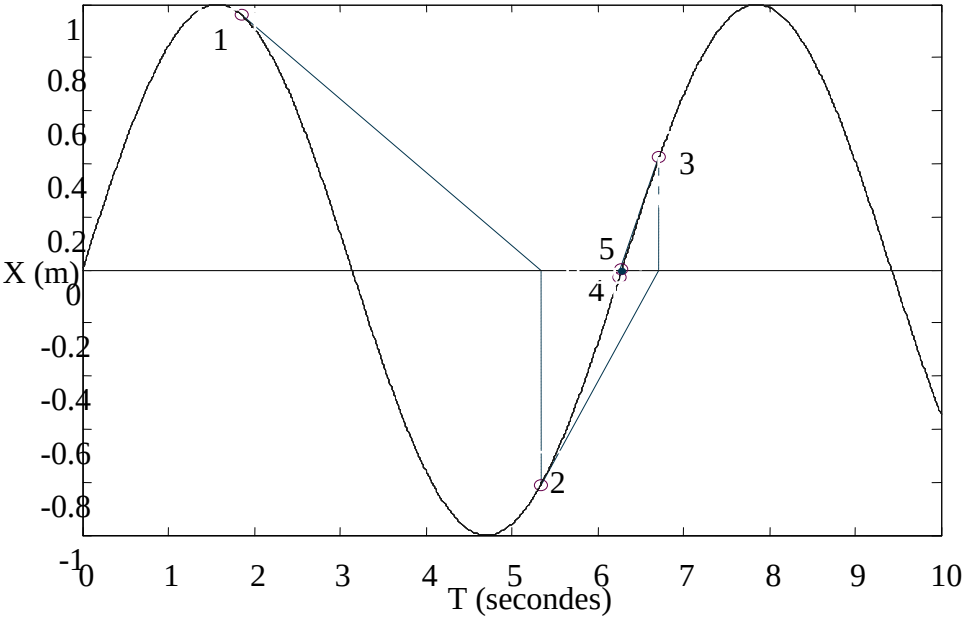


Figure 5 : Convergence de la méthode de Newton-Raphson à une racine qui n'est pas la plus proche sur une fonction sinusoïdale

2.2.3.2 Méthodes en n-dimensions

2.2.3.2.1 Méthode de Newton-Raphson en n-dimensions

Si les méthodes pour calculer les racines de fonctions à une variable sont simples et offrent de très hauts taux de succès, il en est tout autre pour leur généralisation en n dimensions ou n variables. Il est, en effet, impossible d'encadrer avec certitude une racine ou une discontinuité comme avec la méthode de la bissectrice. La méthode de Newton-Raphson peut cependant être généralisée à n-dimensions. L'algorithme décrit plus haut devient donc :

1. Calculer $\vec{y}_0, \vec{f}(\vec{y}_0)$ et $J(\vec{y}_0)$
2. Tant que $\|\vec{y}_{i+1} - \vec{y}_i\| > tol$

$$\vec{y}_{i+1} = \vec{y}_i - J(\vec{y}_i)^{-1} * \vec{f}(\vec{y}_i)$$

Où l'indice i est l'itération et $J(\vec{y}_i)$ est le Jacobien calculé au point d'opération $\vec{y}_i$, soit la matrice des dérivées partielles des fonctions d'état par rapport aux variables d'état :

$$J = \begin{bmatrix} \frac{df_1}{dy_1} & \cdots & \frac{df_1}{dy_n} \\ \vdots & \ddots & \vdots \\ \frac{df_n}{dy_1} & \cdots & \frac{df_n}{dy_n} \end{bmatrix} \quad (59)$$

Comme dans l'algorithme en une dimension, le choix de l'estimé initial est crucial. Une bonne connaissance de la position approximative de la racine permet d'augmenter significativement les chances de succès. Il est donc nécessaire d'injecter de l'information à tout algorithme pour l'aider à converger vers la racine.

La méthode de Newton-Raphson consiste donc à linéariser le modèle autour de l'estimé initial. Le modèle peut alors être réécrit comme suit à l'aide d'un développement en séries de Taylor :

$$\vec{f}(\vec{y} + \delta\vec{y}) = \vec{f}(\vec{y}) + J * \delta\vec{y} + O(\delta\vec{y}^2) \quad (60)$$

En négligeant les termes d'ordre supérieur $O(\delta\vec{y}^2)$, il est possible de résoudre le système d'équation suivant :

$$\vec{f}(\vec{y} + \delta\vec{y}) = \vec{f}(\vec{y}) + J * \delta\vec{y} \quad (61)$$

Puisque la racine de ce modèle simplifié est recherchée, le terme $\vec{f}(\vec{y} + \delta\vec{y})$ est posé à zéro. Il ne reste donc qu'à calculer la correction $\delta\vec{y}$ de l'équation :

$$0 = \vec{f}(\vec{y}) + J * \delta\vec{y} \quad (62)$$

qui devient :

$$\delta\vec{y} = -J^{-1} * \vec{f}(\vec{y}) \quad (63)$$

Puisque le modèle original n'est généralement pas linéaire, le processus itératif consiste à mettre les variables d'état y à jour en y ajoutant les corrections $\delta\vec{y}$ successives calculées.

En pratique, des ressources importantes sont nécessaires pour inverser le Jacobien. Cependant, il n'est pas nécessaire d'inverser complètement une matrice pour pouvoir résoudre une équation de la forme de (63).

La méthode la plus utilisée est la décomposition LU qui permet de décomposer A en deux matrices L (matrice diagonale inférieure, L pour « lower ») et U (matrice diagonale supérieure, U pour « upper »). En termes de ressources, il faut $\frac{1}{3}N^3$ opérations (addition, soustraction, multiplication, division) pour factoriser une matrice et N^2 opérations supplémentaires pour résoudre le système d'équation présenté en (63) pour un total de $N^2 + \frac{1}{3}N^3$ opérations (Press et al. 1994). Ainsi, la solution à l'équation de (63) sera donnée par l'équation (64):

$$\delta\vec{y} = L^{-1}\left(U^{-1} * \vec{f}\right) \quad (64)$$

Puisque L et U sont des matrices triangulaires, leur solutions successives sont calculées directement sans nécessiter d'inversion. (Press et al. 1994)

Cette décomposition est en fait si répandue que l'écriture sous la forme de (63) relève de l'abus de langage puisque presqu'aucun algorithme n'inverse complètement une matrice

(ce qui requière N^3 opérations) avant de résoudre le système d'équation associé. La très grande majorité se contentera de factoriser (ou décomposer) A sous une forme plus facile à résoudre comme la décomposition LU. Cependant, lorsqu'une matrice doit être résolue à répétition avec de légères modifications, une décomposition dite QR est généralement plus avantageuse. Cette décomposition, plutôt que de décomposer J en une matrice triangulaire inférieure et une matrice triangulaire supérieure, décompose J en une matrice triangulaire supérieure (R) et une matrice orthogonale (Q). Bien que la décomposition requière environ le double d'opération de la décomposition LU, la forme des sous-matrices permet leur mise-à-jour en O(N2) opérations, c'est-à-dire un nombre d'opérations indéterminé, mais dont l'ordre de grandeur est donné par le carré de la taille de la matrice. Le Jacobien décomposé peut donc être mis à jour très rapidement sur la base du nouveau point d'opération sans avoir à le recalculer en entier ni à le redécomposer.

2.2.3.2.2 Méthode de Broyden

Bien que la méthode de Newton-Raphson en N-Dimensions puisse être appliquée sous la forme présentée, le premier algorithme efficace de résolution de système non-linéaire a été développé par (Broyden 1965). Il se base sur les méthodes de Newton-Raphson tout en augmentant la stabilité de l'algorithme. L'implémentation la plus couramment utilisée est tirée de (Press et al. 1994). Il s'agit de l'algorithme utilisé sur la plateforme Tornado. De plus, la méthode de Broyden ne nécessite pas de Jacobien exact. Elle se contente d'une approximation construite par différences finies

$$\frac{d\vec{f}}{d\vec{y}} \cong \frac{f(\vec{y}_2) - \vec{f}(\vec{y}_1)}{\vec{y}_2 - \vec{y}_1} \tag{65}$$

L'avantage d'utiliser une approximation réside dans le fait que les modèles non-linéaires comportent souvent beaucoup d'équations. Ces équations n'étant pas toujours dérivable analytiquement, par exemple en présence de fonctions non-analytiques comme des conditions (IF-ELSE) ou autre, la construction d'un Jacobien symbolique est généralement une tâche ardue. En outre, très peu de logiciels permettent une dérivation symbolique. De plus, la construction d'un Jacobien par différences finies est très rapide numériquement

puisque le modèle doit simplement être calculé N+1 fois alors que le calcul de chaque terme du Jacobien peut mener à un nombre d'opération beaucoup plus important.

La méthode de Broyden calcule ses itérations de façon similaire à la méthode de Newton-Raphson. La principale différence réside dans l'utilisation d'un Jacobien numérique. De plus, le Jacobien est décomposé sous forme QR, plus efficace que la décomposition LU puisque les itérations successives modifient le modèle suffisamment lentement pour que le Jacobien puisse être mis à jour plutôt que recalculé entièrement et redécomposé.

Ainsi, l'algorithme tel que proposé par (Broyden 1965) et implémenté par (Press et al. 1994) est présenté à la Figure 6.

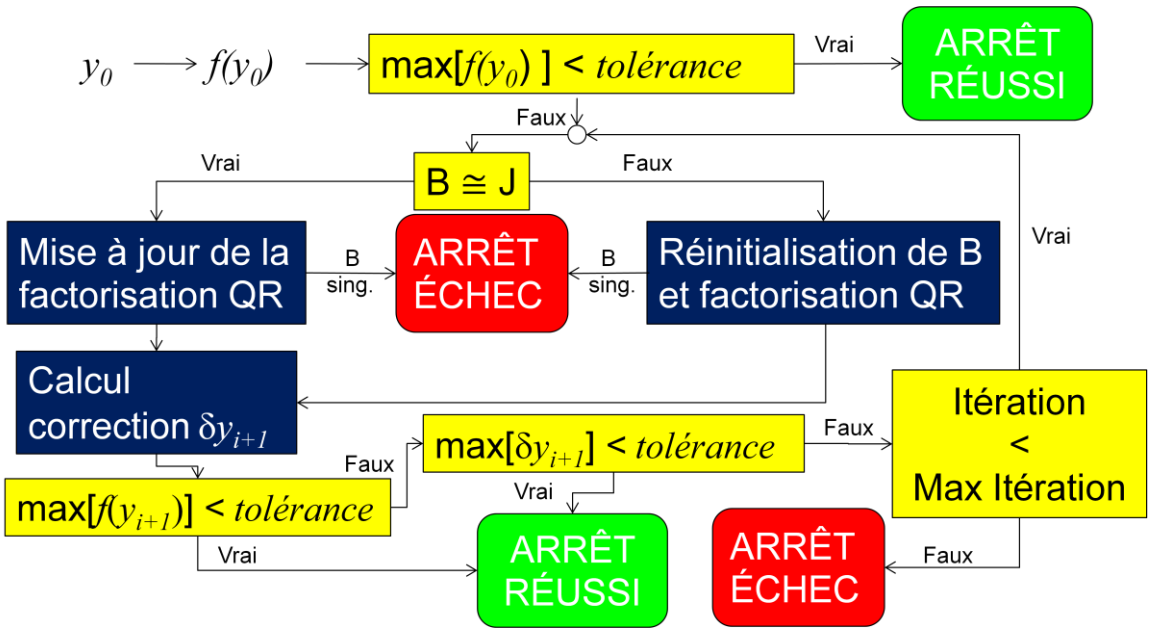


Figure 6: Algorithme de Broyden pour la résolution de système d'équations non-linéaires tel qu'implémenté en Tornado

Ainsi, l'algorithme proposé réussit à trouver une racine si l'estimé initial est une racine (solution triviale, mais qui évite d'avoir à calculer le Jacobien). Dans le cas contraire, une approximation du Jacobien par différences finies (notée B dans le schéma) est utilisée pour calculer des corrections successives à l'estimé initial. Si la matrice B est singulière (ce qui empêche de calculer la correction $\delta\vec{y}_{i+1}$), l'algorithme retourne un message d'erreur et la racine (si elle existe) ne peut être calculée.

Pour conclure, l'algorithme de Broyden se base sur une recherche de racine dans la direction de descente maximale indiquée par l'approximation du Jacobien. Cette stratégie a fait ses preuves sur des problèmes présentant une faible non-linéarité. Sur des problèmes plus complexes comme les modèles mathématiques de traitement biologique d'eaux usées, des discontinuités ou autres événements très non-linéaires peuvent empêcher la convergence vers une solution. Selon l'expérience acquise au cours de ce projet, aucun modèle biologique n'a pu être résolu à l'aide de l'algorithme de Broyden, ce qui rend d'autant plus nécessaire le second algorithme disponible suivant.

2.2.3.2.3 Algorithme Hybride de la bibliothèque d'algorithme MINPACK

Le second algorithme disponible est tiré de la bibliothèque MINPACK. Bien que très semblable à l'algorithme de Broyden, l'algorithme Hybride utilise la puissance de la méthode de Powell et de la méthode de la Région de Confiance (Moré et al. 1980).

La méthode de Powell (Powell 1962) repose sur l'idée générale de la direction de recherche de la plus grande pente. En une dimension, cette idée revient à la méthode de Newton où la direction de recherche est simplement la direction dans laquelle la variable se rapproche du zéro. Lorsque le nombre de dimensions (ou de variables) augmente, les algorithmes numériques recherchent habituellement la direction où la pente est maximale (c'est le cas, notamment, de l'algorithme de Broyden). Cette direction est donnée par la solution de l'équation (63). Il reste ensuite à l'algorithme à trouver la distance optimale à parcourir dans cette direction. Si cette stratégie offre d'excellents résultats sur une ou deux itérations, il existe très peu de cas où elle s'avère idéale. Le cas classique en deux dimensions est présenté à la Figure 7.

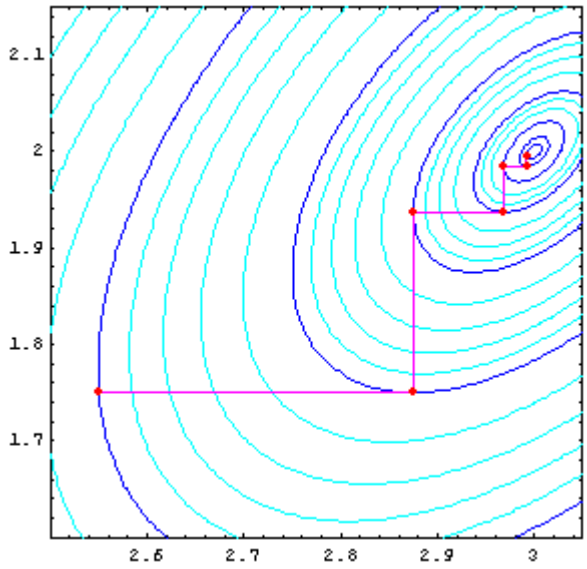


Figure 7: Illustration de la méthode de la pente descendante maximale[1]

Là où Powell innove, c'est dans la gestion des directions de recherches successives. Il suffit, selon lui, de calculer d'abord la correction dans la direction de descente maximale durant deux itérations. Connaissant ces deux directions, la troisième est calculée comme étant la somme des deux premières. La Figure 8 montre l'effet de la méthode.

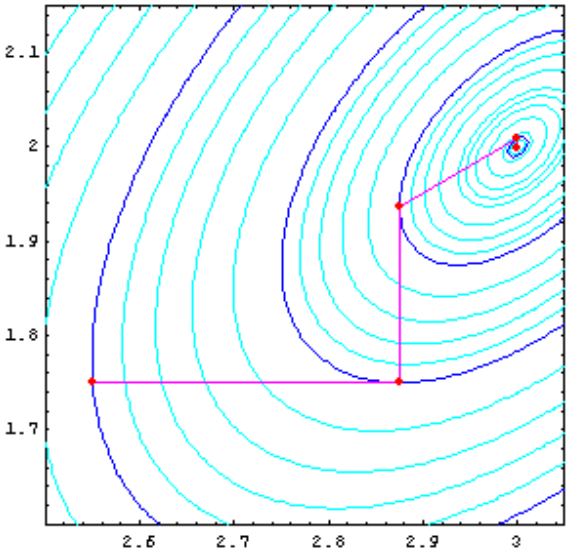


Figure 8: Illustration de la méthode de Powell pour la recherche d'une racine[2]

[1] http://math.fullerton.edu/mathews/n2003/PowellMethodMod.html - 27 février 2009

Si le pas est jugé satisfaisant, la direction d'un des deux premiers pas est remplacée par celle de cette nouvelle direction et le processus recommence jusqu'à ce que la tolérance soit atteinte. Cette méthode est aussi connue sous le nom de « dogleg » ou méthode du zigzag (Powell 1962).

La méthode de la zone de confiance propose une approche alternative à la méthode de la pente descendante maximale. Lorsque le Jacobien est utilisé pour calculer une direction de recherche, le modèle est dit linéarisé, c'est-à-dire qu'autour du point considéré, les dérivées de toutes les variables sont considérées constantes. Cette approche permet de calculer facilement une approximation de la racine. Cependant, cette approximation n'est évidemment pas valide sur toute la plage de variation des variables. La zone de confiance vient donner une frontière à la correction maximale qui pourra être apportée pour se diriger vers la racine. Le pas dans la direction de recherche doit donc minimiser le terme :

$$\left\|\vec{f} + J * \delta\vec{y}\right\| \tag{66}$$

tout en respectant la contrainte :

$$\|D * \delta\vec{y}\| \leq \Delta \tag{67}$$

Ainsi, la correction $\delta\vec{y}$ doit minimiser l'ensemble d'équations $\vec{f}(\vec{y}, t)$. Cependant, la norme du produit de la correction et d'une matrice d'échelle D doit rester inférieure à une constante Δ. Tant qu'une correction satisfaisante n'est pas trouvée, Δ est réduit. Une fois la correction optimale trouvée et appliquée aux variables d'état, D, Δ et le Jacobien (si nécessaire) sont mis à jour et une nouvelle itération est lancée jusqu'à ce que la solution converge vers le régime permanent (Moré et al. 1980).

2.2.4 Algorithmes d'Optimisation

Les algorithmes de résolution d'équations non-linéaires se déclinent généralement en deux implémentations, soit une implémentation recherchant les zéros de N équations algébriques

---

[2] http://math.fullerton.edu/mathews/n2003/PowellMethodMod.html - 27 février 2009

non-linéaires simultanément et une implémentation cherchant à minimiser une fonction objectif donnée par l'utilisateur.

La première implémentation revient à la résolution d'équations non-linéaires et renvoie un code d'erreur si les équations ne peuvent obtenir de valeur nulle en un seul point. La seconde implémentation correspond à une autre catégorie d'algorithme : les optimiseurs. La méthode de Newton-Raphson possède une version permettant d'optimiser détaillée par (Edgar et al. 2001). La bibliothèque MINPACK offre également une version de son algorithme Hybride en mode d'optimisation. Ceci étant dit, les deux versions ne poursuivent pas les mêmes objectifs. Néanmoins, les recherches réalisées sur les algorithmes d'optimisation ne doivent pas être passées sous silence puisqu'une transformation dans le domaine de la résolution d'équations non-linéaires peut être possible.

Il est possible de rechercher la racine d'un ensemble d'équations directement à l'aide d'un algorithme d'optimisation. Il suffit de lui donner comme critère à minimiser la somme au carré des fonctions d'état (voir équation (68)).

$$crit = \sum f_i^2 \qquad (68)$$

Ainsi, lorsque le critère d'arrêt $crit$ est nul ou sous une certaine tolérance, une racine est identifiée.

Malheureusement, en règle générale, cette stratégie n'offre pas de résultats probants. La principale raison donnée par (Press et al. 1994) est qu'un algorithme d'optimisation ne fait pas de différence entre un minimum local et un minimum global (sauf quelques algorithmes d'exception exigeant de très longs temps de calcul comme les algorithmes génétiques, de Recuit Simulé (« Simulated Annealing ») ou encore de colonies de Fourmies (« Ant Colony Optimization » ACO). Un minimum local donnant un critère d'arrêt non-nul est synonyme d'échec de l'algorithme. Donc à moins de lancer plusieurs optimisations en séries selon des stratégies comme celle de Monte-Carlo, les chances sont grandes que l'algorithme s'arrête sur un minimum local qui n'est pas une racine.

2.2.5 Traitement de contraintes

Dans le domaine de la modélisation des stations d'épuration, le modélisateur dispose généralement de connaissances externes au modèle. L'exemple le plus simple d'une telle connaissance est la fourchette dans laquelle devraient varier les variables d'état. L'ajout de contraintes à un modèle doit permettre d'utiliser cette information pour favoriser la convergence d'un algorithme purement mathématique vers les solutions ayant un sens physique.

2.2.5.1 Fonction de pénalité

Une fonction de pénalité peut être intégrée à des algorithmes d'optimisation pour simuler des contraintes d'égalité (Edgar et al. 2001). L'idée de base est de transformer un problème d'optimisation avec contrainte en un problème équivalent sans contrainte.

Problème avec contrainte :

$$\begin{aligned} \vec{y}' &= \vec{f}(\vec{y}) \\ 0 &= \vec{g}(\vec{y}) \end{aligned} \tag{69}$$

Problème sans contrainte :

$$\vec{P}(\vec{y}, r) = \vec{f}(\vec{y}) + r * \left(\vec{g}(\vec{y})\right)^2 \tag{70}$$

Dans le problème avec contrainte, le paramètre r est le paramètre de pénalité. Puisque la fonction $\vec{g}(\vec{y})$ est strictement positive, la fonction $\vec{P}(\vec{y}, r)$ trouvera son minimum là où $\vec{g}(\vec{y})$ est près de zéro. Un tel problème est généralement résolu de manière itérative en augmentant graduellement la valeur de r. En effet, plus r est petit, moins la contrainte n'est considérée. À l'inverse, plus r est élevé, plus la contrainte est contraignante.

L'avantage d'une méthode de pénalité est de pouvoir lancer un problème avec contraintes sur des algorithmes déjà bien connus et maîtrisés travaillant uniquement sur des modèles sans contrainte. L'inconvénient, par contre, vient de la structure du problème reformulé. En effet, un tel problème génère une matrice hessienne ($\nabla^2 \vec{P}$) très mal conditionnée lorsque r devient très grand (Edgar et al. 2001). Une matrice mal conditionnée est numériquement

difficile à résoudre compte tenu des variations importantes des ordres de grandeur des nombres. Les erreurs numériques peuvent alors s'accumuler et devenir très importantes. Une telle technique doit donc être utilisée avec circonspection.

2.2.5.2 Fonction de barrière

L'ajout d'une barrière est similaire à une fonction de pénalité, mais s'applique à des contraintes d'inégalité (Edgar et al. 2001). Un problème d'optimisation avec contrainte d'inégalité est donc transformé en un problème sans contrainte comme suit :

Problème avec contrainte :

$$\begin{matrix} \vec{y}' = \vec{f}(\vec{y}) \\ 0 \leq \vec{g}(\vec{y}) \end{matrix} \tag{71}$$

Problème sans contrainte :

$$\vec{P}(\vec{y}, r) = \vec{f}(\vec{y}) - r * \ln\left(\vec{g}(\vec{y})\right) \tag{72}$$

Puisque la contrainte $\vec{g}(\vec{y})$ ne peut être négative, l'essai d'une valeur proche de la contrainte retournera un terme positif à $\vec{P}(\vec{y}, r)$ tendant vers l'infini, ce qui pénalisera la valeur essayée. Encore une fois, le paramètre r permet de travailler itérativement avec une valeur faible pour commencer et une valeur qui augmente pour ramener la valeur optimale du bon côté de la contrainte.

Il faut cependant noter que dans le cas de la fonction de pénalité comme dans le cas de la barrière, la valeur dite optimale varie en fonction du paramètre r. La solution trouvée n'est donc pas exacte, mais tend vers la solution exacte puisque les équations à traiter sont modifiées fortement.

2.2.5.3 Transformation des variables

La transformation de variable est une opération purement mathématique qui consiste à changer l'intervalle sur lequel rechercher un optimum pour éliminer les contraintes sur cette variable (Dochain et Vanrolleghem 2001). Ainsi, une variable contrainte entre un minimum et un maximum :

$$y_{min} < y < y_{max} \tag{73}$$

peut être transformée pour remplacer y par l'expression équivalente en ϕ donnée par l'équation (74) :

$$\phi = \tan\left(\frac{\pi}{2} * \frac{2y - y_{max} - y_{min}}{y_{max} - y_{min}}\right) \tag{74}$$

L'avantage réside en le fait que ϕ peut varier sans contrainte et être retransformé en y en tout temps sans jamais enfreindre les limites sur y, tel qu'illustré sur la Figure 9:

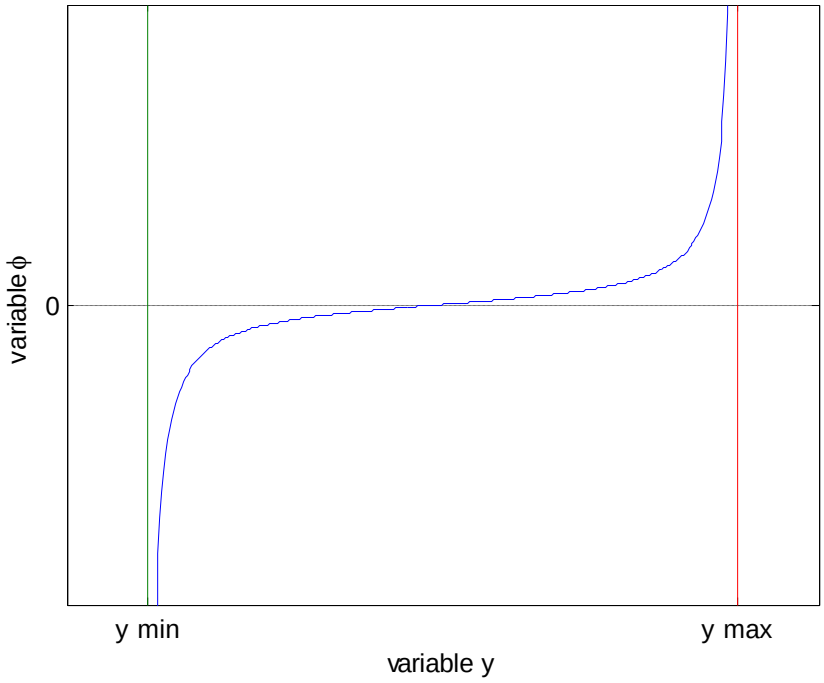


Figure 9 : Équivalence entre la variable y et ϕ.

Par la suite une optimisation sur ϕ peut être réalisée sans contrainte. Lorsque l'optimum est trouvé, il ne reste qu'à recalculer la valeur de y à l'aide de l'équation (75) :

$$y = \frac{1}{2} * (y_{max} + y_{min}) + (y_{max} + y_{min}) * \frac{\arctan(\phi)}{\pi} \tag{75}$$

Cette technique possède un avantage très important par rapport aux fonctions de pénalité et de barrière. Il s'agit de l'absence de perturbation des fonctions d'état. Ainsi, d'un point de

vue algébrique, il n'existe aucune différence entre les variables y et ϕ puisque la transformation pour passer de l'une à l'autre est toujours exacte. De plus, travailler avec ϕ plutôt qu'avec une fonction de pénalité ou de barrière ne requiert qu'une seule itération puisqu'aucun paramètre ne doit être modifié pour augmenter la précision de la réponse.

2.2.6 Algorithmes alternatifs

Les algorithmes étudiés jusqu'à maintenant sont couramment utilisés en pratique. Cependant, il existe une multitude de modèles ne pouvant être résolus par ces méthodes. C'est pourquoi plusieurs modifications permettant de favoriser la convergence sur des problèmes très complexes ont été développées.

2.2.6.1 Algorithme utilisant le concept d'Homotopie

Une variante intéressante de l'algorithme Hybride est l'œuvre de (Paloschi 1994). Dans ses travaux, il a démontré que l'utilisation du concept d'homotopie pouvait améliorer la convergence significativement dans certains problèmes que l'algorithme Hybride était incapable de résoudre.

Le concept d'homotopie consiste à prendre deux objets mathématiques et à les transformer de l'un à l'autre de façon continue. Dans le cadre d'un algorithme de régime permanent, ces deux objets sont deux ensembles d'équations et la transformation est donnée par la formule suivante :

$$H(\vec{y}, \Theta) = (1 - \Theta) * \vec{f}(\vec{y}) + \Theta * \vec{g}(\vec{y}) \qquad (76)$$

Ainsi, le paramètre Θ variera entre 0 et 1 pour offrir une transformation continue entre $\vec{f}(\vec{y})$ et $\vec{g}(\vec{y})$. Tout ce qui reste à déterminer est la forme que prendra le système d'équations $\vec{g}(\vec{y})$.

La méthode proposée par (Paloschi 1994) consiste à d'abord tenter de résoudre le système d'équation $\vec{f}(\vec{y})$ à l'aide de l'algorithme Hybride. En cas d'échec, un système d'équation $\vec{g}(\vec{y})$ est construit sous la forme :

$$\vec{g}(\vec{y}) = \vec{f}(\vec{y}) - \vec{f}(\vec{y}_k) \quad (77)$$

où $\vec{f}(\vec{y}_k)$ est la meilleure solution calculée par l'algorithme Hybride. Ce système est facile à résoudre numériquement lorsque $\vec{y}$ est proche de $\vec{y}_k$. Le paramètre Θ prend alors une valeur différente de zéro. Plus cette valeur est grande, plus l'influence du système d'équation original est faible et plus le système résultant est facile à résoudre. Lorsqu'une solution au système d'équation $H(\vec{y}, \Theta)$ est trouvée, il est possible de réduire Θ pour trouver une solution du système d'équation original, ce qui est fait lorsque Θ est ramené à zéro.

Bien que cette méthode semble présenter un excellent taux de succès (98% des problèmes testés par les auteurs ont convergé à une solution contre 61% lorsque l'algorithme Hybride était utilisé seul), elle est également très coûteuse en temps de calcul puisque chaque variation du paramètre Θ entraîne la solution d'un nouveau système d'équation.

2.2.6.2 Couplage simulation – résolution de système d'équation pour calculer le Régime Permanent

Certains modèles sont si complexes que ni la simulation, ni les algorithmes traditionnels pour calculer le Régime permanent ne sont efficaces. C'est le cas présenté par (Shiraishi et al. 2009) dans le cadre d'un modèle de voies métaboliques. Le modèle décrit est très raide (« stiff »), ce qui nécessite des algorithmes particuliers pour le résoudre en simulation. Ces algorithmes, très spécialisés, n'étaient pas suffisant puisque certaines variables d'état devenaient négatives. Lorsque ces variables représentent des quantités comme des concentrations, une valeur négative est synonyme d'échec de l'algorithme.

Pour résoudre leur problème, (Shiraishi et al. 2009) proposent de transformer graduellement leur système d'équations différentielles ordinaires (ÉDO ou ODE en anglais) en système d'équation différentielles et algébriques (ÉDA ou DAE en anglais) pour calculer le régime permanent du modèle.

Une méthode empirique est donc proposée qui consiste à d'abord tenter de résoudre le système d'équations différentielles par un algorithme traditionnel. En cas d'échec, toutes les équations différentielles ayant retourné une valeur finale négative sont transformées en

équations algébriques implicites (donc de la forme $\vec{y}' = \vec{f}(\vec{y})$ à la forme $0 = \vec{f}(\vec{y})$) et une nouvelle solution est calculée.

Il est à noter que les résultats dynamiques ne sont pas exacts et ne tentent pas de représenter la réalité. Seules les valeurs au régime permanent sont d'intérêt.

Comme dans le cas de la méthode utilisant le concept d'Homotopie, il s'agit d'un algorithme à n'employer que lorsque les algorithmes traditionnels sont inefficaces compte tenu de la lourde surcharge de calcul impliquée.

2.3 Sommaire

Beaucoup de travaux ont été réalisés sur des techniques numériques pour favoriser l'utilisation de modèles mathématiques. De ces travaux, deux approches sont observables. La première consiste à offrir une solution robuste et performante à tous les problèmes mathématiques. Les algorithmes développés ici seront d'une très grande simplicité et un minimum de superflu est visible. Ainsi, l'algorithme d'intégration RK4 est toujours utilisé bien que d'une très grande simplicité. En régime permanent, les algorithmes de Broyden et Hybride sont deux implémentations de la méthode de Newton-Raphson où le calcul du Jacobien est simplifié à l'aide de différences finies. La stabilité de ces algorithmes est également améliorée pour permettre la résolution de plus de problèmes qu'un simple algorithme de Newton-Raphson (méthode de la zone de confiance, par exemple). Mais dans l'ensemble, il s'agit d'algorithmes réduits à leur plus simple expression.

La seconde approche consiste à développer des méthodes spécifiques à des problèmes difficiles. Ainsi, les algorithmes d'intégration capable de résoudre des modèles raides (algorithme de Runge-Kutta implicite, CVODE, technique de différentiation arrière) sont très performants mais ce sont également des algorithmes très complexes qui pourront prendre plus de ressources pour résoudre des problèmes simples. Dans la même veine, utiliser une méthode d'homotopie pour résoudre un modèle au régime permanent offre d'excellents taux de convergence. Par contre, une telle approche implique la résolution d'un grand nombre de sous-modèles avant de calculer la solution à un modèle final. Le temps de calcul peut donc exploser et devenir problématique.

3 Matériel et Méthodes

3.1 Tornado

L'environnement de développement de ce mémoire est Tornado, développé par MOSTforWATER (Kortrijk, Belgique). Il s'agit d'un ensemble d'outils mathématiques permettant de réaliser des expériences virtuelles (évaluations de modèles mathématiques). Son architecture logicielle est abondamment documentée dans (Claeys 2008a).

Les outils disponibles incluent un compilateur (MOF2T) permettant de transformer un modèle écrit dans un langage de modélisation (Modelica ou MSL) en un modèle exécutable.

Tornado est développé en C++, un langage de programmation orienté objet. Son développement modulaire permet la superposition de nombreux algorithmes sous forme de bibliothèque de liens dynamiques (dynamic link library ou dll en anglais). Le développement d'un nouvel algorithme se fait donc en créant une nouvelle bibliothèque sur le modèle d'un algorithme existant.

L'environnement Tornado offre également des fonctions permettant de communiquer avec certaines fonctions d'un modèle, selon les besoins. Les modèles à résoudre étant généralement très complexes, une seule fonction permettant d'évaluer les équations différentielles est insuffisante. C'est pourquoi, les fonctions suivantes doivent être appelées selon le besoin :

```
m_pCallbackTime->SetTime(t);
m_pInput->Input();
m_pModel->ComputeState();
m_pModel->ComputeOutput();
m_pModel->CheckBounds();
m_pModel->Update();
m_pOutput->Output();
```

La première fonction (`SetTime(t)`) sert à indiquer à Tornado où en est rendue la simulation. La seconde (`Input()`) sert à calculer les entrées provenant d'un fichier d'entrées ou d'un générateur d'entrées du modèle. La fonction `ComputeState()` représente le cœur du modèle. C'est dans cette fonction que sont calculées les équations différentielles

sur la base des variables d'état. La fonction `ComputeOutput()` est appelée à chaque pas de temps complété pour calculer les variables en sortie de l'algorithme. La fonction `CheckBounds()` s'assure que les limites ne sont pas dépassées. Le comportement de cette fonction doit être spécifié par l'utilisateur. Ainsi, en cas de dépassement des limites, il est possible de : ne rien faire, émettre un avertissement tout en continuant la simulation et émettre un message d'erreur et arrêter l'expérience. La fonction `Update()` est utilisée pour gérer les mémoires tampons nécessaires à certaines fonctions. Finalement, la fonction `Output()` gère les sorties, que ce soit pour les enregistrer dans un fichier texte, pour tracer un graphique ou pour communiquer avec une application extérieure à Tornado.

3.1.1 Algorithmes disponibles – Intégrateurs

La plateforme Tornado possède 25 implémentations d'intégrateurs différents. Cependant, cela ne signifie pas qu'ils sont tous utilisés en pratique. En effet, six de ces intégrateurs sont des implémentations particulières de la méthode de Runge-Kutta à pas fixe, c'est-à-dire que l'erreur accumulée à chaque pas n'est pas considérée.

Néanmoins, la présence de nombreux algorithmes différents permet de choisir celui le plus adapté au calcul d'un modèle particulier. (Claeys 2008b) propose d'ailleurs différentes avenues pour optimiser le choix d'un algorithme.

Il faut également noter que dans sa version actuelle, la plateforme Tornado ne peut résoudre que des modèles composés d'équations différentielles ordinaires :

$$\vec{y}' = \vec{f}(\vec{y}, t) \tag{78}$$

Il n'est donc pas possible de résoudre des modèles décrits par des systèmes d'équations différentielles et algébriques :

$$\begin{aligned} \vec{y}' &= \vec{f}(\vec{y}, \vec{z}, t) \\ 0 &= \vec{g}(\vec{y}, \vec{z}, t) \end{aligned} \tag{79}$$

3.1.2 Algorithmes disponibles – Résolution de systèmes d'équations

Si plusieurs intégrateurs sont disponibles, il en va tout autrement dans le cas des algorithmes de résolution de systèmes d'équations non-linéaires nécessaires pour, par exemple, calculer le régime permanent d'un modèle. Les deux seules implémentations disponibles sont la méthode de Broyden et l'algorithme Hybride de la bibliothèque MINPACK.

Dans leur implémentation actuelle, les deux algorithmes utilisent une décomposition QR pour inverser leur Jacobien. Cependant, seul l'algorithme de Broyden envoie un message d'erreur si le Jacobien est singulier, bien que la décomposition soit réussie.

Bien qu'un Jacobien singulier soit synonyme de système d'équation sous-déterminé (autrement dit, moins d'équations indépendantes que de variables), il existe des cas de figure où un tel comportement est recherché. C'est le cas lorsqu'une équation algébrique implicite sert de contrainte sous la forme :

$$0 = h(\vec{y}, \vec{z}, t) \tag{80}$$

Pour travailler, l'algorithme essaie des valeurs différentes de $\vec{y}$ et de $\vec{z}$ jusqu'à ce que la fonction soit annulée. Cependant, durant tout le processus, le résidu de la fonction doit être assigné à une variable. L'équation (80) doit donc être reformulée comme suit :

$$z_{n+1} = g_{n+1}(\vec{y}, z_0, \dots, z_n, t) \tag{81}$$

Sous cette forme, puisque z_{n+1} n'apparaît dans aucune équation, toutes les dérivées par rapport à elle-même seront nulles. La colonne $\frac{d}{dz_{n+1}}$ rend donc tout Jacobien singulier. De plus, une décomposition LU telle qu'implémentée par (Press et al. 1994) impliquerait une division par zéro, ce qui interdirait l'utilisation de la méthode. Par contre, la décomposition QR peut être réalisée sur une matrice singulière et une solution à la matrice peut être calculée. Seulement, une matrice singulière étant sous-déterminée, les variables comme z_{n+1} de l'équation (81) peuvent prendre n'importe quelle valeur et résoudre l'équation. Leur comportement sur certains algorithmes peut donc être aléatoire.

3.1.3 Expériences virtuelles

La plateforme Tornado permet de réaliser diverses expériences virtuelles. Le tutoriel joint à l'Annexe A permet de réaliser les diverses expériences disponibles.

Ces expériences possèdent différents niveaux hiérarchiques. Les deux expériences de base sont le calcul du régime permanent (`ExpSSRoot`, `ExpSSOptim`) et la simulation dynamique (`ExpSimul`). Les expériences de niveau supérieur utilisent ces expériences de base pour construire des structures répondant à divers objectifs. La Figure 10 est tirée de (Claeys 2008a) et montre les différents niveaux d'expériences.

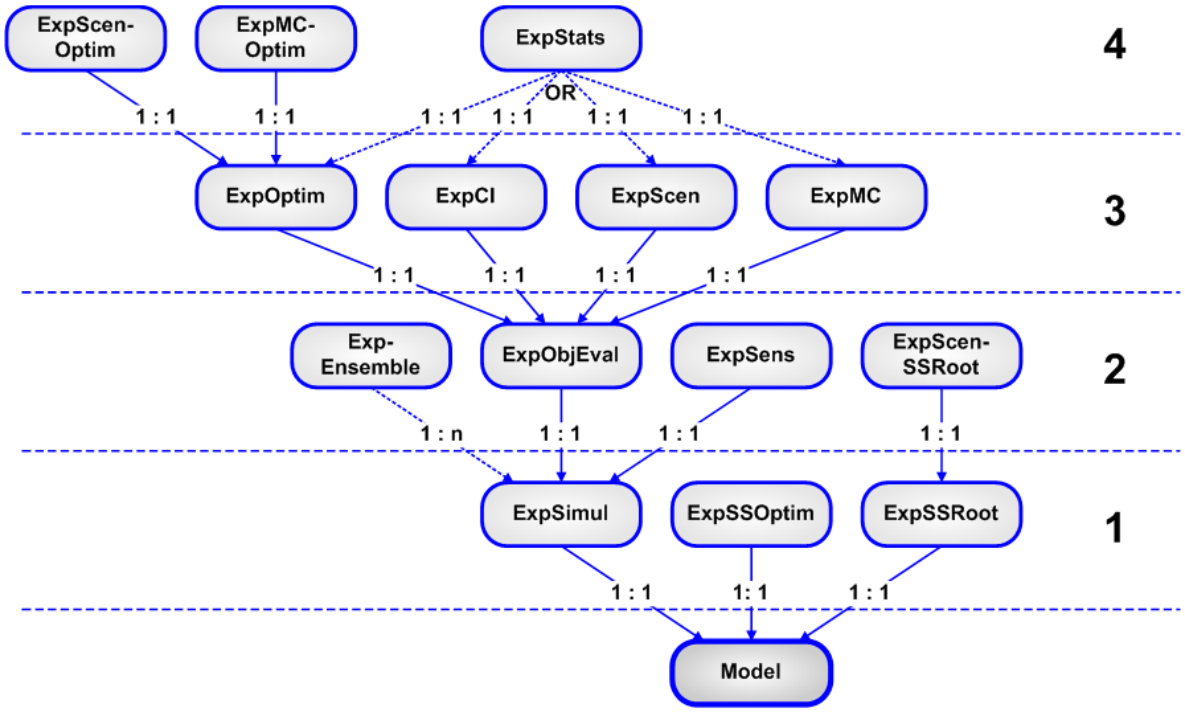


Figure 10 : Différentes expériences virtuelles disponibles sur la plateforme Tornado (Claeys 2008a).

Dans le cadre de ce projet, seules les expériences de simulation (`ExpSimul`) et de calcul du régime permanent (`ExpSSRoot`) ont été utilisées.

Comme le montre la Figure 10, l'élément essentiel pour lancer une simulation est le modèle en soit. Les modèles utilisés en Tornado sont typiquement écrits en Modelica ou en MSL. Il est également possible de créer un modèle sur le logiciel WEST. Le logiciel WEST est un logiciel de simulation également développé par MOSTforWATER. Dans un futur proche,

WEST et Tornado fusionneront pour que le premier offre une interface graphique conviviale et efficace alors que Tornado gérera les aspects du calcul numérique. Il est déjà possible de convertir un modèle d'une plateforme à l'autre, offrant une possibilité supplémentaire pour le développement de modèles.

Puisque les fichiers contenant le modèle sont de simples fichiers textes, la première étape consiste à construire une bibliothèque de liens dynamiques (fichier .dll) à partir du fichier texte. Pour se faire, deux applications développées sur la plateforme Tornado sont nécessaires. La première, `MOF2T.EXE`, sert à compiler le modèle une première fois et à le convertir en langage C. Cette étape permet également d'optimiser le modèle en réalisant certaines simplifications et en triant efficacement les équations du modèle (Claeys et al. 2007) (voir la section Compilateur MOF2T). Une fois l'exécution de `MOF2T.EXE` terminée, deux fichiers sont créés. Le premier est le modèle comme tel en langage C et le second est un fichier xml contenant toutes les informations symboliques (liens entre les paramètres, valeurs initiales des paramètres et variables, etc.).

La seconde application est `TBUILD.EXE`. Son rôle est de générer la bibliothèque de liens dynamiques sur la base du modèle réécrit en C.

Lorsque le modèle est généré et fonctionnel, il est possible de créer les expériences virtuelles. Une expérience virtuelle se présente sous la forme d'un fichier xml. Ce fichier est généré automatiquement par l'application `TMAIN.EXE` de Tornado et contient toutes les informations dont l'exécuteur d'expérience `TEXEC.EXE` a besoin pour réaliser l'expérience virtuelle en question. Un fichier représentant une simulation est présenté ci-dessous :

```
<Tornado>
  <Exp Version="1.0" Type="Simul">
    <Props>
      <Prop Name="Author" Value="PCYRIL\Cyril"/>
      <Prop Name="Date" Value="Mon Feb 09 15:46:01 2009"/>
      <Prop Name="FileName"
Value="AcidBase.Simul.Exp.xml|.Tornado"/>
      <.../>
    </Props>
    <Simul>
      <Model Name="AcidBase" CheckBounds="false"
             StopWhenBoundsViolation="false"
             StopWhenSteadyState="false">
      </Model>
      <Inputs Enabled="true">
        <Input Name="*Calc*">
        </Input>
      </Inputs>
      <Outputs Enabled="true">
        <Output Name="File">
          <File Name="AcidBase.Simul.out.txt" Enabled="true">
            <Props>
              <Prop Name="CommInt" Value="0.01"/>
              <Prop Name="Precision" Value="8"/>
              <.../>
            </Props>
          </File>
        </Output>
      </Outputs>
      <Time>
        <Props>
          <Prop Name="StartTime" Value="0"/>
          <Prop Name="StopTime" Value="50"/>
        </Props>
      </Time>
      <Solve>
        <Integ Method="CVODE">
          <Props>
            <Prop Name="MaxNoSteps" Value="0"/>
            <Prop Name="RelativeTolerance" Value="1e-0015"/>
            <.../>
          </Props>
        </Integ>
      </Solve>
    </Simul>
  </Exp>
</Tornado>
```

Il est possible de modifier ce fichier selon les besoins à remplir puisque l'expérience virtuelle a préséance sur le modèle symbolique généré à la compilation. Si un paramètre se voit assigner une valeur dans l'expérience symbolique, la valeur assignée dans le modèle symbolique est écrasée. Cette gestion permet un accès facile aux paramètres d'intérêt sans

modifier la version originale du modèle puisqu'il est toujours possible d'y revenir en effaçant les assignations du fichier de l'expérience.

Une fois l'expérience virtuelle bien définie, il ne reste qu'à l'exécuter à l'aide de l'application `TEXEC.EXE`. Cette application fait le lien entre les algorithmes choisis pour réaliser l'expérience et le modèle.

L'Annexe A décrit les différentes expériences virtuelles disponibles ainsi que les manipulations à réaliser en ligne de commande (DOS) pour y parvenir.

3.1.4 Compilateur MOF2T

Tel qu'indiqué précédemment, la plateforme Tornado supporte quelques langages de programmation (Modelica, MSL, etc.). Cependant, pour être exécuté, un modèle doit être compilé en une bibliothèque de liens dynamiques. Puisqu'un compilateur, commercial (ex : Visual Studio de Microsoft) ou non (ex : Borland), doit être utilisé pour générer cette bibliothèque, un programme est nécessaire pour convertir les langages de modélisation en un langage plus commun, le C. Sur la plateforme Tornado, ce programme est MOF2T.

MOF2T lit un modèle écrit en Modelica et l'interprète selon les préceptes de lex & yacc (Levine et al. 1992). Le programme réalise d'abord une analyse lexicale. Durant cette étape, le modèle est lu et converti en une séquence d'unités lexicales. Ces unités lexicales, aussi appelées des nœuds, représentent chaque élément du modèle et un nœud spécifique existe pour chaque élément possible. Cette interprétation s'exprime mieux par un exemple. Ainsi, l'équation (82) est réécrite sous forme de ses différents nœuds à la Figure 11.

$$x = 3 * y + (\sin(\pi) - 2) \qquad (82)$$

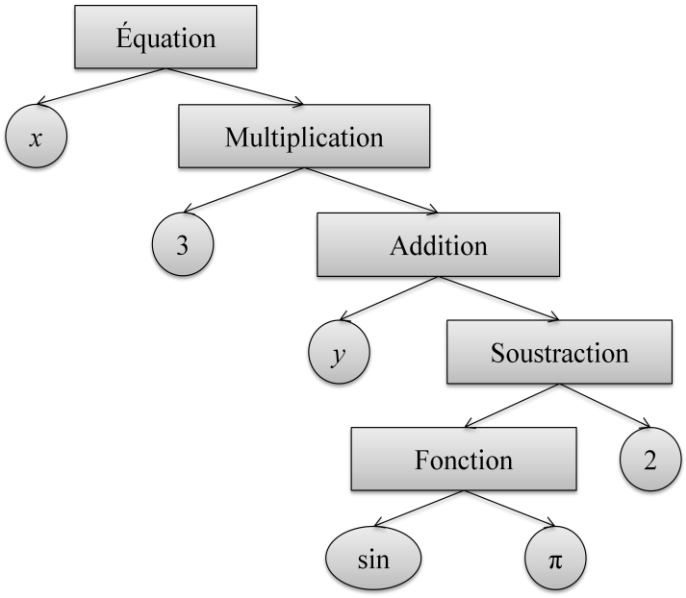


Figure 11 : représentation de l'équation (82) sous forme d'unités lexicales

Cette forme de représentation permet de réécrire tout le modèle sous la forme d'un arbre d'unités où les différentes opérations sont liées. De plus, la représentation de la Figure 11 intègre la gestion de la priorité des opérations. En effet, le nœud « Multiplication » multiplie 3 au résultat du nœud Addition. Il est donc possible de descendre l'arbre pour connaître les opérations à réaliser et d'ensuite le remonter en remplaçant chaque nœud par la solution de son expression.

Le programme MOF2T représente donc le modèle en entier de cette façon en commençant avec un nœud appelé FCLASS qui indique le début du modèle. Ce nœud est ensuite lié à différents autres nœuds qui composent les différentes sections d'un modèle. Il est donc possible de parcourir le modèle en entier en passant d'un nœud à l'autre. C'est ainsi que le modèle est simplifié. Trois opérations sont réalisées sur les équations du modèle. D'abord, les expressions constantes sont évaluées. Ainsi l'expression $(\sin(\pi) - 2)$ de l'équation (82) est remplacée par sa valeur (-2) durant cette étape. Ensuite, les équivalences sont résolues. Ces équivalences sont de la forme $x = y$. En remplaçant les x par le y correspondant, les temps de calculs ont montré des réductions allant jusqu'à 30% (Claeys 2008a). Finalement certaines équations ne sont basées que sur des paramètres et des constantes. Ces équations retournent donc une valeur constante tout au long d'une simulation. MOF2T s'assure alors de

ne les calculer qu'une seule fois en les déplaçant à la section des équations initiales. Le même raisonnement est fait sur les équations ne servant qu'à calculer les sorties. Ces équations sont regroupées dans une section spécifique qui ne sera calculée que lorsque les variables en sortie sont demandées. Un gain de temps de calcul allant jusqu'à 20% a pu être observé par (Claeys 2008a).

Représenter les équations sous forme d'arbre permet finalement l'utilisation d'équations récursives. Par exemple, une fonction servant à substituer toutes les occurrences de y dans un modèle par z commencerait par lire le premier nœud de l'équation. S'il s'agit de la variable y, la substitution est réalisée et la fonction est terminée. Dans le cas contraire, si le nœud n'a pas d'enfant (des liens vers d'autres nœuds), il n'y a pas de substitution et la fonction est terminée. Si le nœud a des enfants (ex : un nœud de multiplication), la fonction est réappelée sur tous les enfants du nœud.

Ces équations récursives tirent leur puissance de leur simplicité. En effet, la fonction de l'exemple précédent ne doit gérer que les trois cas décrits et ce, peu importe la complexité des équations. Écrire une fonction équivalente sans utiliser de fonction récursive nécessiterait l'usage de boucles de programmation complexes et lourdes.

3.2 Modèles utilisés pour tester les algorithmes

Un algorithme d'intégration ou de calcul du régime permanent est un outil. Comme tel, il est donc incapable de retourner de résultat si aucun modèle ne lui est fourni, de la même façon qu'il est impossible de savoir si un marteau va bien sans clou à enfoncer ou à retirer. C'est dans ce but que trois modèles ont été utilisés pour déterminer les performances des algorithmes proposés. Ils sont tous développés en Modelica puisque c'est ce langage qu'accepte le logiciel Tornado.

3.2.1 Modèle Acide – Base

L'objectif de ce modèle est de représenter les réactions d'équilibre des espèces suivantes :

$$NH3 + H^+ \rightleftharpoons NH_4^+ \tag{83}$$

$$OH^- + H^+ \rightleftharpoons H_2O \tag{84}$$

Ces équations d'équilibre décrivent les quatre demi-réactions présentées ci-dessous:

$$NH_3 + H^+ \overset{k_1}{\rightarrow} NH_4^+ \tag{85}$$

$$NH_4^+ \overset{k_2}{\rightarrow} NH_3 + H^+ \tag{86}$$

Et

$$OH^- + H^+ \overset{k_3}{\rightarrow} H_2O \tag{87}$$

$$H_2O \overset{k_4}{\rightarrow} OH^- + H^+ \tag{88}$$

Les constantes cinétiques peuvent être réécrites sous la forme mieux connue suivante (Reichert et al. 2001) :

$$\begin{aligned} k\_N &= k_1 \\ K\_eq\_N &= \frac{k_2}{k_1} \\ k_w &= k_3 \\ K\_eq\_w &= \frac{k_4}{k_3 \times H_2O} \end{aligned} \tag{89}$$

Dans le contexte de cette étude, cependant, les cinétiques revêtent une importance minimale puisque seul l'équilibre est recherché. Cependant, le fait de pouvoir les modifier permet de changer radicalement le comportement du modèle.

3.2.1.1 Modèle en langage Modelica

```
fclass AcidBase
  parameter Real k_w = 1e-4;
  parameter Real k_N = 1e-1;
  parameter Real K_eq_w = 1e-14;
  parameter Real K_eq_N = 10^(-9.2);
  parameter Real N_tot = 1e-5;
  parameter Real pH_init = 7;

  Real S_H(start = 1e-7, min = 0);
  Real S_NH3(start = 0, min = 0);
  Real S_NH4(start = 1e-5, min = 0);
  Real S_OH(start = 1e-7, min = 0);
  Real Z_plus(start = 1e-5);
  Real delta_Z(start = 0);

  output Real o_S_H(min = 0);
  output Real o_S_OH(min = 0);
  output Real o_S_NH3(min = 0);
  output Real o_S_NH4(min = 0);
  output Real o_delta_Z;

equation
  der(S_OH) = k_w * (1 - (S_H * S_OH) / K_eq_w);
  der(S_H) = k_w * (1 - S_H * S_OH / K_eq_w)
           + k_N * (S_NH4 - S_NH3 * S_H / K_eq_N);
  der(S_NH3) = k_N * (S_NH4 - S_NH3 * S_H / K_eq_N);
  S_NH4 = N_tot - S_NH3;
  der(delta_Z) = (S_H + S_NH4 - S_OH - Z_plus);

  o_S_H = S_H;
  o_S_OH = S_OH;
  o_S_NH3 = S_NH3;
  o_S_NH4 = S_NH4;
  o_delta_Z = delta_Z;
end AcidBase;
```

La première section d'un modèle écrit en Modelica représente la déclaration des variables et paramètres. C'est ici que sont déclarés les paramètres `k_w`, `k_N`, `K_eq_N` et `K_eq_w`. Le paramètre `N_tot` représente la quantité d'azote totale dans le modèle donnée par :

$$N_{tot} = NH_3 + {NH_4}^+ \tag{90}$$

Les variables `S_H`, `S_NH3`, `S_NH4` et `S_OH` représentent les concentrations des ions et molécules H^+, NH_3, NH_4^+ et OH^-. Le préfixe « `S_` » vient de la notation utilisée dans les modèles de traitement d'eaux usées désignant les espèces solubles par « `S` » et les espèces particulaires par « `X` ». Finalement, la variable `Z_plus` représente la charge résiduelle en

solution et les ions spectateurs. Sa valeur est calculée sur la base des concentrations initiales de H^+, NH_4^+ et OH^- comme suit :

$$Z_{plus} = H^+ + NH_4^+ - OH^- \quad (91)$$

La variable `delta_Z` est une variable fictive. Elle sert à ajouter une équation algébrique de la forme

$$0 = g(y, t) \quad (92)$$

Le but étant de générer l'équation (93) .

$$0 = H^+ + NH_4^+ - OH^- + Z_{plus} \quad (93)$$

Cette forme d'équation n'est pas permise en Modelica – Tornado. Cependant, lorsque le régime permanent est calculé, il est possible d'ajouter cette équation sous forme d'équation différentielle, d'où le besoin d'une variable dont la valeur devra être zéro. De plus, sa présence ne devrait pas influencer le calcul du régime dynamique du modèle. Malheureusement, l'équation (93) sous forme différentielle (équation (94)) est numériquement instable. Lorsque le modèle est résolu en régime dynamique, cette équation doit donc être éliminée.

$$\frac{dDelta_Z}{dt} = H^+ + NH_4^+ - OH^- + Z_{plus} \quad (94)$$

La section des variables de sorties (dont la déclaration commence par « output ») sert à extraire les valeurs lors de l'exécution du modèle.

3.2.1.2 Équations Initiales

La section d'équations initiales, non présente dans le modèle présenté, sert à initialiser des paramètres ou les valeurs initiales des variables d'état. Cependant, les techniques employées par Tornado interdisent d'initialiser les variables d'état dans cette section si le régime permanent doit être calculé puisque les équations de cette section sont calculées à

chaque itération lorsque le régime permanent est recherché. Néanmoins, les valeurs initiales des différentes variables sont calculées manuellement à l'aide des équations suivantes :

$$S\_H = 10^{-pH_{initial}} \tag{95}$$

$$S\_OH = \frac{K\_eq\_w}{S\_H} \tag{96}$$

$$S\_NH4 = N\_tot - S\_NH3 \tag{97}$$

$$Z\_plus = S\_H + N\_tot - S\_NH3 - S\_OH \tag{98}$$

3.2.1.3 Équations d'État

Ce modèle Acide-Base contient trois variables d'état indépendantes : H^+, OH^- et NH_3. La concentration du NH_4^+ n'est pas indépendante puisqu'elle peut être calculée algébriquement à partir d'un simple bilan de masse sur l'azote. La première équation différentielle contrôle la concentration en ion hydroxyde :

```
der(S_OH) = k_w * (1 - (S_H * S_OH) / K_eq_w);
```

La seconde équation différentielle contrôle la concentration en ions hydrogènes :

```
der(S_H) = k_w * (1 - S_H * S_OH / K_eq_w)
           + k_N * (S_NH4 - S_NH3 * S_H / K_eq_N);
```

Cette équation dépend de l'équilibre $H^+ - OH^-$ et de l'équilibre $NH_3 - NH_4^+$. La troisième équation différentielle contrôle la concentration d'ammoniac :

```
der(S_NH3) = k_N * (S_NH4 - S_NH3 * S_H / K_eq_N);
```

L'équation suivante représente la relation entre l'ammoniac et l'ion ammonium. Puisque la quantité d'azote dans le modèle est constante, la concentration de l'ion ammonium est calculée directement :

```
S_NH4 = N_tot - S_NH3;
```

En observant les trois équations différentielles de plus près, il est possible de voir que l'équation contrôlant H^+ est une combinaison linéaire des deux autres. Si ce fait ne crée aucun problème lors de simulations dynamiques, le système ne permet pas de résoudre simultanément les trois équations en régime permanent puisqu'il est sous-déterminé, c'est-à-dire qu'il possède une infinité de solutions. Ce système sous-déterminé est résolu par

l'ajout d'une quatrième équation contenant une variable fictive. Cette équation sert à fixer l'équilibre de la charge du modèle Acide – Base :

$$0 = H^+ + NH_4^+ - OH^- + Z^+ \quad (99)$$

Puisque la plateforme Tornado n'accepte pas d'équation implicite, l'équation (99) doit être réécrite sous une forme explicite d'une équation différentielle au régime permanent possédant une variable fictive, donc sans intérêt tant que l'équation est résolue :

```
der(delta_Z) = (S_H + S_NH4 - S_OH - Z_plus);
```

Le modèle sera toujours sous-déterminé, mais la seule variable qui soit possible de fixer arbitrairement devient cette variable fictive delta_Z, garantissant que le sous-système d'intérêt est, lui, bien déterminé.

3.2.1.4 Équations de sortie

Finalement, les quatre dernières équations permettent de générer des fichiers de sortie du modèle dynamique.

```
o_S_H = S_H;
o_S_OH = S_OH;
o_S_NH3 = S_NH3;
o_S_NH4 = S_NH4;
o_delta_Z = delta_Z;
```

3.2.2 Modèle Prédateur – Proie

Le second modèle utilisé pour tester les algorithmes est un modèle suivant les populations de prédateurs et de proies. Ce modèle est tiré de la base de modèles de Tornado et est dérivé du modèle classique de Lotka-Volterra. Il ne contient que deux variables d'état, soit les populations de prédateur et de proie. Il tient compte des processus les plus simples des modèles prédateur-proie, soit les naissances – mortalité de chaque population et la prédation. Ses équations d'état sont :

$$\frac{dProie}{dt} = c4 * Proie - (c1 * Pred * Proie + c5 * Proie^2) \quad (100)$$

$$\frac{dPred}{dt} = b * c1 * Pred * Proie - (c2 * Pred + c3 * Pred^2) \quad (101)$$

Dans ces conditions, le modèle Prédateur – Proie peut exprimer quatre comportements dynamiques qui dépendent des populations initiales des proies et prédateurs :

1. Les deux populations sont à zéro : Le modèle ne montre aucune évolution des populations et restent à zéro.
2. La population de proies est nulle : La population de prédateur décroit jusqu'à zéro.
3. La population de prédateur est nulle : La population de proies croît jusqu'à ce que l'équilibre entre les naissances et les décès soit atteint.
4. Les deux populations sont non-nulles : Un équilibre est atteint entre les deux populations.

Les paramètres utilisés (Tableau 2) ne font également pas apparaître de cycle entre les populations de prédateurs et de proies. Le modèle converge donc toujours vers une solution stable.

Tableau 2 : Paramètres du modèle Prédateur – Proie

Paramètre	Valeur
c1	0.001
c2	0.9
c3	0.0001
c4	1.1
c5	1e-5
b	0.02

3.2.2.1 Modèle en langage Modelica

```
fclass PredatorPrey
  parameter Real c1 = 0.001;
  parameter Real c2 = 0.9;
  parameter Real c3 = 0.0001;
  parameter Real c4 = 1.1;
  parameter Real c5 = 1e-5;
  parameter Real b = 0.02;

  output Real pa_out;
  output Real ps_out;

  input Real pa.in_1;
  input Real pa.in_2;
  output Real pa.out_1;
  Real pa.p(start = 1000);

  input Real ps.in_1;
  input Real ps.in_2;
  output Real ps.out_1;
  Real ps.p(start = 100);

  input Real c5papa.in_1;
  input Real c5papa.in_2;
  output Real c5papa.out_1;
  parameter Real c5papa.c = 1.0;

  input Real c1paps.in_1;
  input Real c1paps.in_2;
  output Real c1paps.out_1;
  parameter Real c1paps.c = 1.0;

  input Real c3psps.in_1;
  input Real c3psps.in_2;
  output Real c3psps.out_1;
  parameter Real c3psps.c = 1.0;

  input Real c4pa.in_1;
  output Real c4pa.out_1;
  parameter Real c4pa.c = 1.0;

  input Real bc1paps.in_1;
  output Real bc1paps.out_1;
  parameter Real bc1paps.c = 1.0;

  input Real c2ps.in_1;
  output Real c2ps.out_1;
  parameter Real c2ps.c = 1.0;

  input Real c2ps_plus_c3psps.in_1;
  input Real c2ps_plus_c3psps.in_2;
  output Real c2ps_plus_c3psps.out_1;

  input Real c5papa_plus_c1paps.in_1;
```

```
  input Real c5papa_plus_c1paps.in_2;
  output Real c5papa_plus_c1paps.out_1;

initial equation
  c1paps.c = c1;
  c2ps.c = c2;
  c3psps.c = c3;
  c4pa.c = c4;
  c5papa.c = c5;
  bc1paps.c = b;

equation
  der(pa.p) = (pa.in_1 - pa.in_2);
  pa.out_1 = pa.in_1 - pa.in_2;
  der(ps.p) = (ps.in_1 - ps.in_2);
  ps.out_1 = ps.in_1 - ps.in_2;
  c5papa.out_1 = c5papa.c * c5papa.in_1 * c5papa.in_2;
  c1paps.out_1 = c1paps.c * c1paps.in_1 * c1paps.in_2;
  c3psps.out_1 = c3psps.c * c3psps.in_1 * c3psps.in_2;
  c4pa.out_1 = c4pa.c * c4pa.in_1;
  bc1paps.out_1 = bc1paps.c * bc1paps.in_1;
  c2ps.out_1 = c2ps.c * c2ps.in_1;
  c2ps_plus_c3psps.out_1 = c2ps_plus_c3psps.in_1 +
                           c2ps_plus_c3psps.in_2;
  c5papa_plus_c1paps.out_1 = c5papa_plus_c1paps.in_1 +
                             c5papa_plus_c1paps.in_2;
  pa_out = pa.p;
  ps_out = ps.p;
  bc1paps.in_1 = c5papa_plus_c1paps.in_1;
  c5papa_plus_c1paps.in_1 = c1paps.out_1;
  c2ps.in_1 = c3psps.in_1;
  c3psps.in_1 = c3psps.in_2;
  c3psps.in_2 = c1paps.in_1;
  c1paps.in_1 = ps.p;
  c2ps_plus_c3psps.in_1 = c2ps.out_1;
  c2ps_plus_c3psps.in_2 = c3psps.out_1;
  ps.in_2 = c2ps_plus_c3psps.out_1;
  ps.in_1 = bc1paps.out_1;
  c1paps.in_2 = c5papa.in_1;
  c5papa.in_1 = c5papa.in_2;
  c5papa.in_2 = c4pa.in_1;
  c4pa.in_1 = pa.p;
  c5papa_plus_c1paps.in_2 = c5papa.out_1;
  pa.in_2 = c5papa_plus_c1paps.out_1;
  pa.in_1 = c4pa.out_1;
end PredatorPrey;
```

3.2.3 Modèle ASM1

Le troisième modèle fut choisi pour avoir un cas classique venant du domaine du traitement biologique des eaux usées. C'est pourquoi un modèle simple d'un réacteur à boues activées aéré ASM1 (Henze et al. 1987) est utilisé à titre de troisième modèle test. Ce modèle a été choisi pour sa complexité, puisqu'il contient 14 variables d'état (soit 11 de plus que le modèle Acide – Base), et pour sa simplicité, puisqu'il ne contient que 14 variables d'état (comparativement à 108 pour le modèle BSM1). Le modèle a été construit dans le logiciel WEST selon la Figure 12.

Figure 12 : Modèle ASM1 construit sur le logiciel WEST

Il faut noter que ce modèle ne se veut pas un modèle de station d'épuration réaliste. Le but de ce modèle est de servir de test pour les différents algorithmes et non de permettre le dimensionnement d'une usine réelle. Ainsi, les valeurs par défaut sont utilisées aussi souvent que possible.

Les caractéristiques de l'eau à traiter sont constantes dans le temps et sont fournies par un générateur de variables d'entrées appelé « Input Generator » dans l'environnement WEST. Le modèle du réacteur aéré est basé sur le modèle ASM1 et comporte un volume fixe.

La version en langage Modelica de ce modèle est trop longue pour être insérée dans ce rapport. Cependant, les paramètres utilisés pour exécuter les expériences virtuelles sont indiquées au Tableau 3 et les valeurs données en entrée au modèle sont au Tableau 4. Il faut noter que le logiciel WEST travaille avec des masses, et non des concentrations. De plus, les entrées sont fournies sous forme de concentration avant d'être converties en masses.

Finalement, le régime permanent a été calculé à l'aide d'une simulation exécutée sur une période de temps suffisamment longue pour que les variables d'état soient stables. Les valeurs du Tableau 5 constituent le régime permanent de référence du modèle.

Puisqu'un algorithme de recherche du régime permanent requiert des estimés initiaux, les variables d'état se voient assigner un estimé initial par défaut. Lorsqu'une recherche des régimes permanents est lancée, seuls deux variables d'état voient leurs estimés initiaux être modifiés. Cette approche offre de bons résultats puisque la variation de deux variables montre les limites des algorithmes de recherche du régime permanent en permettant de détecter des solutions à l'équilibre très variées. Les estimés initiaux de base sont affichés au Tableau 6.

Tableau 3 : Valeur numérique des paramètres du modèle ASM1

Paramètre	Valeur	unités
F_BOD_COD	0,65	–
F_TSS_COD	0,75	–
K_NH	1	$\frac{g\ NH_3 - N}{m^3}$
K_NO	0,5	$\frac{g\ NO_3 - N}{m^3}$
K_OA	0,4	$\frac{g\ O_2}{m^3}$
K_OH	0,2	$\frac{g\ O_2}{m^3}$
K_S	20	$\frac{g\ DCO}{m^3}$
K_X	0,02	$\frac{g\ DCO\ lentement\ biodégradable}{g\ DCO\ cellulaire}$
Kla	50	$jour^{-1}$
S_O_Sat	8	$\frac{g\ O_2}{m^3}$
Y_A	0,24	$\frac{g\ DCO\ produite}{g\ N\ consommée}$
Y_H	0,67	$\frac{g\ DCO\ produite}{g\ DCO\ consommée}$
b_A	0,01	$jour^{-1}$
b_H	0,4	$jour^{-1}$
f_P	0,08	–
i_X_B	0,086	$\frac{g\ N}{g\ DCO}$
i_X_P	0,06	$\frac{g\ N}{g\ DCO}$
k_a	0,06	$\frac{m^3}{g\ DCO \times jour}$
k_h	2	$\frac{g\ DCO\ lentement\ biodégradable}{g\ DCO\ cellulaire \times jour}$
mu_A	0,55	$jour^{-1}$
mu_H	4	$jour^{-1}$
n_g	0,8	–
n_h	0,4	–

Tableau 4 : Valeurs données en entrée au modèle ASM1 et Décanteur Ponctuel

Variable	Valeur	Unité
Mean[H2O]	1 000	$m^3/jour$
Mean[S_ALK]	10	g/m^3
Mean[S_I]	10	g/m^3
Mean[S_ND]	1	g/m^3
Mean[S_NH]	5	g/m^3
Mean[S_NO]	5	g/m^3
Mean[S_O]	2	g/m^3
Mean[S_S]	100	g/m^3
Mean[X_BA]	10	g/m^3
Mean[X_BH]	10	g/m^3
Mean[X_I]	10	g/m^3
Mean[X_ND]	1	g/m^3
Mean[X_P]	1	g/m^3
Mean[X_S]	10	g/m^3

Tableau 5 : Valeur des variables d'état au Régime Permanent calculé par une simulation dynamique

Variable d'état	Valeur au Régime permanent (g)	Valeur au Régime permanent $\left(\frac{g}{m^3}\right)$	Concentrations en entrée $\left(\frac{g}{m^3}\right)$
H2O	1e+009	1e+6	
S_ALK	9 593,46	9,593	10
S_I	10 000	10,000	10
S_ND	824,923	0,825	1
S_NH	49,531	0,050	5
S_NO	5 741,04	5,741	5
S_O	7 001,52	7,002	2
S_S	9 197,03	9,197	100
X_BA	10 147,7	10,148	10
X_BH	67 004	67,004	10
X_I	10 000	10,000	10
X_ND	43,229	0,043 2	1
X_P	3 152,23	3,152	1
X_S	471,74	0,472	10

Tableau 6 : Valeur des estimés initiaux des variables d'état utilisées pour lancer la recherche du régime permanent

Variable d'état	Estimé initial (g)	Estimé initial (g/m^3)
H2O	1e+009	1e+006
S_ALK	9 000	9
S_I	10 000	10
S_ND	800	0,8
S_NH	50	0,05
S_NO	5 000	5
S_O	7 000	7
S_S	9 000	9
X_BA	10 000	10
X_BH	67 000	67
X_I	10 000	10
X_ND	40	0,04
X_P	3 000	3
X_S	400	0,4

3.2.4 Modèle ASU (Activated Sludge Unit)

Le modèle ASU est tiré de la banque de modèles tests fournis sur la plateforme Tornado et sur le logiciel WEST. Il est présenté à la Figure 13.

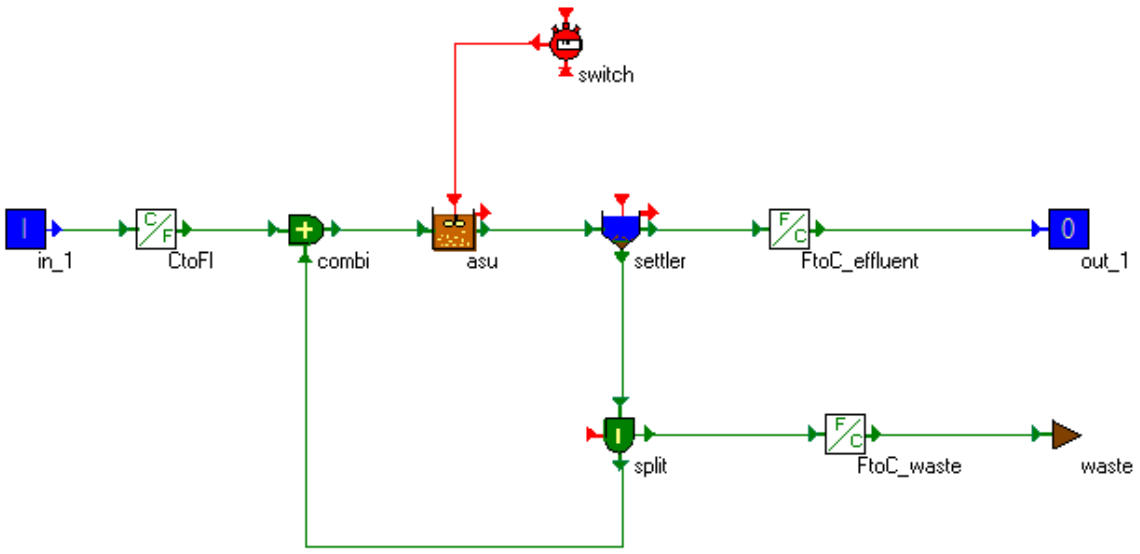


Figure 13 : Représentation sous WEST du modèle ASU

Le réacteur biologique à volume variable est modélisé par le modèle ASM2d qui comporte 20 variables d'état. Le décanteur utilise le modèle de (Takacs et al. 1991) qui subdivise le décanteur en dix couches pour simuler la décantation de la liqueur mixte. Il contient dix variables d'état. La concentration en oxygène est contrôlée par une minuterie (« switch » sur la Figure 13) qui fonctionne sur le mode « ouvert – fermé ». Finalement, un fichier d'entrées standard permet de simuler le modèle ASU en régime dynamique avec un affluent typique de station d'épuration (voir Figure 14) sur sept jours.

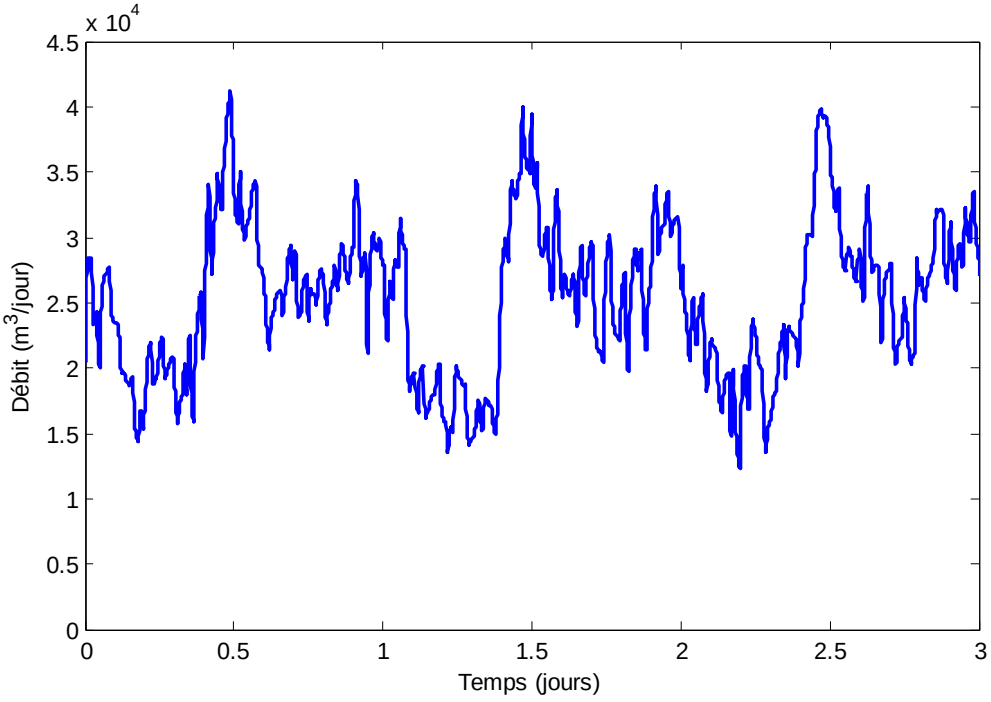


Figure 14 : Profil de débit journalier typique

Tous les paramètres du modèle sont les paramètres par défaut proposés par la plateforme Tornado.

L'utilisation d'une minuterie pour l'aération présente un comportement particulier complexe du point de vue numérique supplémentaire au modèle ASM1, ce qui le rend plus difficile à résoudre en simulation. Le fait d'ajouter un décanteur non-idéal de type Takács et un modèle biologique plus complexe (ASM2d plutôt qu'ASM1 (Henze et al. 2000)) ajoute à la difficulté. Bref, bien que ce modèle ne compte que 30 variables d'état, il

comporte tout de même trois éléments de base de tout modèle biologique de station d'épuration (réacteur aéré, décanteur et contrôleur sous forme de minuterie).

3.2.5 Modèle Benchmark

Ce modèle, également tiré de la banque de modèles tests de Tornado et WEST, se veut une implémentation simple d'un modèle de station d'épuration (Copp 2002). Ainsi, contrairement au modèle ASU, le modèle Benchmark se veut un modèle réaliste d'une station. Il est représenté à la Figure 15.

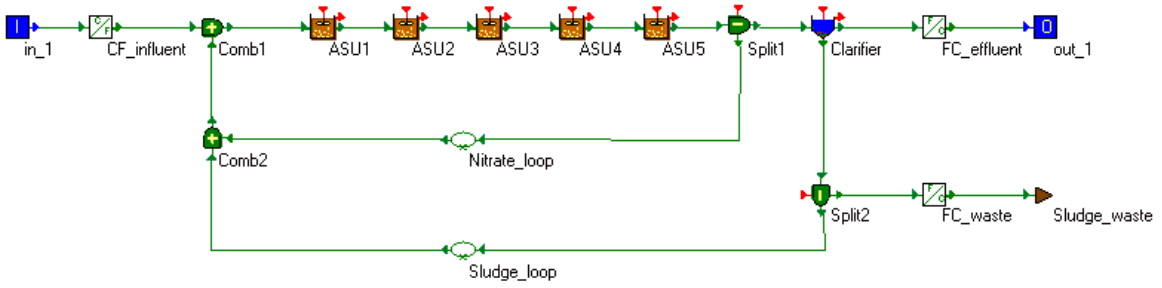


Figure 15 : Représentation sous WEST du modèle Benchmark

Le Benchmark compte donc cinq réacteurs biologiques en série suivis d'une recirculation des nitrates et d'un décanteur secondaire construit sur le modèle de décantation de (Takacs et al. 1991). Le modèle biologique utilisé dans les réacteurs est ASM1 (Henze et al. 1987).

Tous les paramètres du modèle sont les paramètres par défaut proposés par la plateforme Tornado. De plus, comme dans le cas du modèle ASU, un fichier d'entrées standard est proposé, offrant la possibilité de réaliser des simulations en régime dynamique avec des comportements journaliers typiques d'une véritable usine d'épuration.

Ce modèle compte donc 14 variables d'état par réacteur, 10 variables d'état pour le décanteur secondaire et 14 variables d'état dans les briseurs de boucle (« loop-breaker ») de la recirculation des nitrates et la recirculation des boues pour un total de 108 variables d'état.

Bien qu'il représente un petit modèle de station d'épuration, le nombre de variables d'état permet de faire apparaître les problèmes liés aux gros modèles biologiques.

3.3 Technique d'évaluation des algorithmes

3.3.1 Détection du régime permanent

Puisque plusieurs vecteurs d'équilibre devront être comparés aux valeurs du régime permanent de référence (voir Tableau 5), un critère de comparaison simple est utilisé pour déterminer automatiquement si le véritable régime permanent est atteint :

$$FonctionObjectif = \sum_{i=1}^{n} \frac{|y_i - y_{iRP}|}{y_{iRP}} \qquad (102)$$

Ce critère tend vers zéro lorsque le régime permanent recherché (y_{RP}) est atteint. Si d'autres régimes permanents apparaissent, la fonction objectif permet de les détecter puisqu'ils montreront des valeurs numériques distinctes.

3.3.2 Algorithmes du régime permanent

La performance d'un algorithme du régime permanent est évaluée de deux façons. Tout d'abord, par la convergence au résultat souhaité. Il sera vu dans les résultats que les trois modèles tests utilisés pour le calcul du régime permanent (Acide – Base, Prédateur – Proie, ASM1) possèdent plusieurs solutions acceptables au point de vue mathématique, mais incorrectes du point de vue de la réalité. Ainsi, si l'algorithme converge à une solution incorrecte, le résultat est pire que si l'algorithme ne converge pas et renvoie un message d'erreur puisque le second cas sera détecté automatiquement alors que le premier peut passer inaperçu et mener à des calculs erronés par la suite.

Dans un second temps, si plusieurs algorithmes convergent à la solution souhaitée, ils peuvent être triés entre eux en fonction du nombre de fois que le modèle a dû être évalué. Ce nombre est une représentation de l'effort en temps de calcul nécessaire pour converger à la solution.

Les algorithmes de calcul du régime permanent peuvent dépendre très fortement de l'estimé initial de la solution. Ainsi, avant de lancer l'algorithme, ce dernier doit avoir une première estimation du résultat final. C'est à partir de cet estimé qu'il construit sa stratégie de

recherche. Un bon algorithme déterminera bien sûr la bonne solution pour une large plage d'estimés initiaux. La plateforme Tornado offre un type d'expérience virtuelle nommée `ExpScenSSRoot` (pour Expérience de type Scénario de recherche du Régime Permanent par les racines du modèle). Cette expérience virtuelle permet de lancer l'algorithme de recherche du Régime permanent à plusieurs reprises en variant les estimés initiaux selon un schéma prédéfini, que ce soit une variation linéaire entre deux valeurs, une variation logarithmique entre ces valeurs ou encore une suite de valeurs prédéfinies.

3.3.3 Algorithmes d'intégration

Les algorithmes d'intégration doivent d'abord être évalués par rapport à un résultat étalon. Bien que des méthodes statistiques existent pour comparer deux courbes (les séries de Fourier, par exemple, voir (Claeys 2008b)), l'évaluation visuelle reste dans bien des cas l'option la plus efficace. En effet, la comparaison visuelle permet en un seul coup d'œil de déterminer si la solution calculée par deux algorithmes est similaire. Si l'une des courbes diverge, l'identification d'un problème est immédiate.

Compte tenu du grand nombre d'algorithmes d'intégrations disponibles sur la plateforme Tornado et dans la littérature en régime dynamique, deux critères numériques permettent de différencier deux algorithmes présentant une solution acceptable. Le premier est le nombre d'exécution du modèle. Ce critère est habituellement déterminant puisqu'une variation par un facteur 10^6 peut être observée sur un même modèle en utilisant deux algorithmes différents (Claeys 2008b). Il s'agit dans la majorité des cas d'un critère suffisant pour préférer un algorithme à un autre. Dans des cas plus rares, deux algorithmes présenteront un nombre d'évaluation du modèle similaire mais demanderont des temps de calcul pouvant aller du simple au double à cause des calculs intermédiaires au sein de l'algorithme. Le temps de calcul peut donc servir de critère pour évaluer un algorithme. Cependant, ce critère dépend de plusieurs facteurs dont l'ordinateur sur lequel la simulation est exécutée, la plateforme de calcul (ex : Tornado ou Matlab) ou encore le nombre de tâches exécutées en parallèle durant la simulation. Toutes ces sources d'erreurs favorisent l'utilisation du nombre d'exécution du modèle comme critère de base pour quantifier les performances d'un algorithme.

4 Résultats

Ce chapitre se divise en trois parties. La première partie aborde les résultats obtenus avec la réduction automatique de modèle. Puisque cette réduction n'est possible que si elle est couplée à un intégrateur, l'algorithme DIRK développé pour l'occasion est abordé à la section 4.1.1. Suivront les techniques utilisées pour simplifier efficacement le modèle et contrôler la solution en simulation dynamique (sections 4.1.2 à 4.1.4)

La section 4.2 décrit les résultats obtenus en tentant d'améliorer les algorithmes de recherche du régime permanent. Deux approches ont été expérimentées, soit la multiplication des équations d'état par une fonction gaussienne pour favoriser la recherche de solution dans une zone donnée des variables d'état et l'ajout de limites aux variables d'état.

Finalement, compte-tenu de l'importance du Jacobien, qui est utilisé dans l'algorithme DIRK, de même que dans les algorithmes de recherche du régime permanent, des techniques de dérivation symbolique sont proposées à la section 4.3. Plus spécifiquement, des dérivées sont proposées aux fonctions non-analytiques rencontrées couramment dans les modèles.

4.1 Algorithme d'intégration et réduction de la raideur – DIRK

Un algorithme d'intégration basé sur la méthode Diagonale Implicite de Runge-Kutta de (Cameron 1983) a été développé pour implémenter des techniques de réduction de la raideur. Cet algorithme a été choisi comme base des travaux sur la réduction automatique de modèle pour son mode de fonctionnement et ses propriétés numériques. En effet, en tant qu'algorithme implicite, l'algorithme DIRK fait une utilisation intensive du Jacobien du modèle. Puisque le Jacobien représente un coût élevé de calcul, il devient rentable de l'utiliser dans la technique de réduction et dans l'algorithme d'intégration. Mais plus important qu'une économie potentielle de temps de calcul, l'algorithme proposé par (Cameron 1983) permet de calculer la solution de modèle composé d'équations différentielles et algébriques (ÉDA). Puisque le but de la réduction de modèle est de transformer certaines équations différentielles en équations algébriques implicites, un algorithme capable de résoudre ces équations implicites est essentiel. Il n'existe actuellement aucun algorithme sur la plateforme Tornado capable de résoudre des systèmes d'ÉDA. Finalement, les propriétés numériques inhérentes aux algorithmes numériques, que ce soit la stabilité de la solution ou la capacité de l'algorithme à résoudre des systèmes raides devraient aider à former un algorithme performant.

4.1.1 Algorithme DIRK

L'algorithme Diagonale Implicite de Runge-Kutta (DIRK) reprend les équations typiques de Runge-Kutta (28), (29) et (30) :

$$\vec{y}_{n+1} = \vec{y}_n + h\sum_{i=1}^{s} b_i \vec{f}\left(\vec{y}_{n,i}, t_{n,i}\right) \tag{103}$$

$$\vec{y}_{n,i} = \vec{y}_n + h\sum_{j=1}^{i-1} a_{ij} \vec{f}\left(\vec{y}_{n,j}, t_{n,j}\right) + \gamma h \vec{f}\left(\vec{y}_{n,i}, t_{n,i}\right) \tag{104}$$

$$t_{n,i} = t_n + c_i h \tag{105}$$

Les valeurs des paramètres a_{ij}, b_i et c_i sont tirées de (Cameron 1983). La puissance de cet algorithme vient de l'utilisation de paramètres communs pour quatre méthodes implicites

imbriquées, soit à un étage jusqu'à quatre étages. Dans ces conditions, s peut prendre des valeurs successives de 1 à 4 et offrir quatre algorithmes de type DIRK où chaque solution permet de calculer la solution d'ordre supérieur, offrant du même coup des estimés successifs de la solution et permettant d'estimer l'erreur sur la solution. Les paramètres utilisés sont ceux affichés à la section 2.1.1.6.

L'algorithme implémenté utilise un mode dit Ordre Variable, Pas Variable. L'ordre de l'algorithme est géré par une estimation de l'erreur à chaque étape de l'algorithme en comparant la réponse à l'ordre n à la réponse à l'ordre $n+1$. Lorsque l'erreur est sous la tolérance exigée par l'utilisateur, la solution est acceptée. Lorsque l'erreur de l'ordre 3 est supérieure à la tolérance, le pas est rejeté et une solution appropriée est prise (réduction de la taille du pas et/ou mise à jour du Jacobien). Finalement, lorsqu'une solution est acceptée, le pas est mis à jour pour avancer à une vitesse optimale. L'implémentation du code de l'algorithme DIRK sur la plateforme Tornado est très fortement inspirée de l'article de (Cameron 1983).

Tel qu'indiqué dans la revue de littérature, les méthodes de Runge-Kutta implicites sont très stables, ce qui signifie que même à faible tolérance, la solution n'est pas précise, mais diverge très peu de la solution exacte, ayant plutôt tendance à surestimer et sous-estimer la solution à chaque pas. Une telle stabilité peut être vérifiée en comparant l'erreur sur la solution d'un même modèle par deux algorithmes différents. Ainsi, la Figure 16 compare les performances de l'algorithme DIRK avec une tolérance de 10^{-3} à l'algorithme très performant CVODE de la bibliothèque de SUNDIALS utilisant une tolérance relative et absolue de 10^{-3} sur le modèle Benchmark Simulation Model 1 (BSM1).

Ainsi, l'algorithme CVODE montre une déviation de près de 15% après quatorze jours de simulation par rapport à l'algorithme DIRK. Il est possible de confirmer que la dérive est bien le fait de l'algorithme CVODE en recommençant la simulation avec une forte tolérance de 10^{-9}. Ainsi, une figure équivalente à la Figure 16 est tracée avec une tolérance forte sur CVODE à la Figure 17.

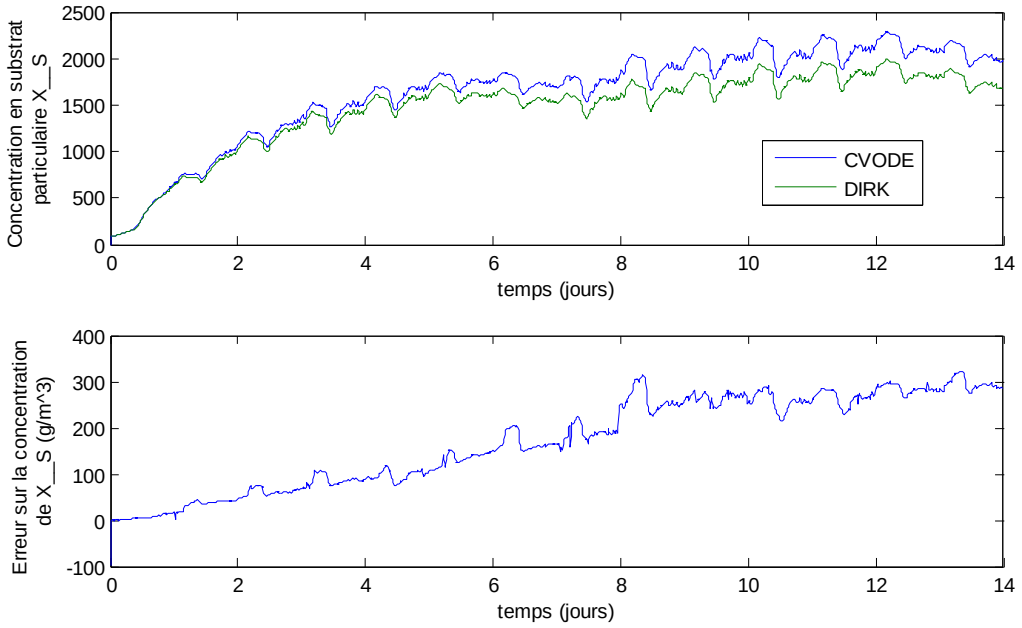


Figure 16 : Comparaison des performances des algorithmes DIRK et CVODE à faible tolérance sur le modèle BSM1

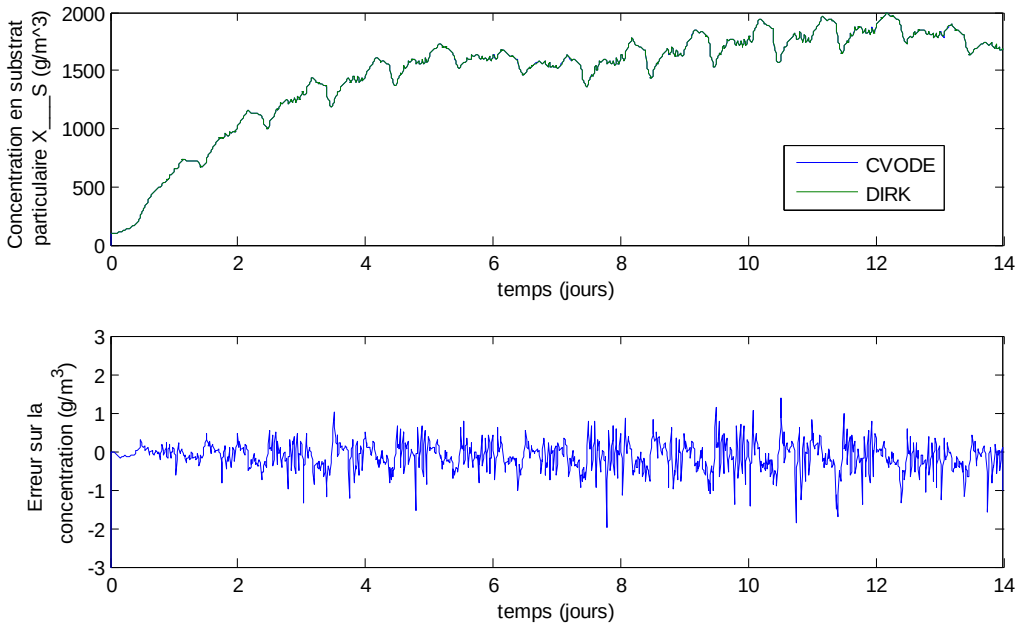


Figure 17 : Comparaison des résultats de l'algorithme DIRK à faible tolérance à l'algorithme CVODE à forte tolérance sur le modèle BSM1.

Tandis que l'erreur augmente si les tolérances sont faibles (Figure 16), la Figure 17 montre que c'est bien l'algorithme CVODE qui diverge. En effet, il est possible de considérer que l'erreur affichée à la Figure 17 est largement due à l'algorithme DIRK puisque sa tolérance est faible. Cependant, l'erreur reste stable dans le temps. Une longue simulation ne devrait donc pas voir apparaître de dérive significative due à l'algorithme DIRK à basse tolérance.

Les résultats de la Figure 16 montrent finalement une utilisation de l'algorithme DIRK, soit le calcul de simulation à faible tolérance. De telles simulations sont souvent nécessaires, que ce soit pour obtenir un estimé peu coûteux de la solution d'un modèle ou pour l'exécution à faible coût d'un nombre élevé de simulations pour une expérience virtuelle (ex : Monte-Carlo ou optimisation). Ces résultats mettent également en garde l'utilisateur contre l'usage de l'algorithme CVODE qui représente l'un des algorithmes les plus performants sur plusieurs modèles. En effet, en demandant une précision de l'ordre de 10^{-3}, un biais atteignant 15% peut être observé. Ainsi, à défaut de savoir régler ses paramètres avec soin (à l'exception des tolérances relatives et absolues), l'utilisation des paramètres par défaut peut mener à des résultats divergeant significativement de la solution exacte. À l'opposée, l'algorithme DIRK démontre une excellente stabilité à faible tolérance.

4.1.2 Réduction automatique d'un modèle

Cette section est tirée en partie de l'article (Garneau et al. 2009).

La réduction automatique de modèle est basée sur les travaux de (Steffens et al. 1997). Elle utilise les valeurs propres du Jacobien du modèle pour trier les variables d'état selon leur constante de temps. Le besoin de réduction de modèle vient du fait que les variables d'état présentant des constantes de temps très courtes sont les principales responsables des temps de calculs très long. Le simple contrôle de la stabilité exige normalement que le pas de temps des intégrateurs explicites (ex : méthode d'Euler, Runge-Kutta 4) soit de l'ordre de grandeur de la constante de temps (voir la section 2.1.1.3 pour une description plus complète de la notion de stabilité des algorithmes). Ainsi, des calculs de pH présents, par exemple, dans les modèles ADM1 (Rosen et al. 2005) ou RWQM1 (Vanrolleghem et al. 2001) reposent sur des constantes de temps de l'ordre de la milliseconde et moins. Réaliser

une simulation dynamique sur ces modèles à l'aide d'un algorithme explicite mène à des temps de calcul prohibitifs. En effet, (Rosen et al. 2006) observent des temps de simulations montrant un facteur de 93 fois le temps de référence simplement en fonction de l'intégrateur et de la forme du modèle.

L'utilisation d'un algorithme implicite implique (DIRK) assure une gestion acceptable de la raideur puisque le pas de calcul sera stable, peu importe sa longueur. Cependant, puisque la longueur du pas est tout de même contrôlée par les variables raides, calculer ces dernières comme étant au régime permanent permettra de franchir des pas plus importants, réduisant d'autant la charge de calcul. C'est pourquoi les variables d'état raides seront calculées comme étant à l'équilibre à chaque pas de temps en posant leur dérivée à zéro.

$$\frac{dy_{raide}}{dt} = 0 \qquad (106)$$

La méthode d'Homotopie présentée à la section 2.1.1.8.3 est donc mise en marche en commençant avec la diagonale du Jacobien. Le lien entre les valeurs propres et les variables d'état est alors automatique puisque chaque élément de la diagonale de la matrice est aussi une valeur propre de cette matrice. Par la suite, une contribution de plus en plus importante du Jacobien est fournie à la matrice selon l'équation (107) où r est le paramètre d'homotopie qui variera de 0 à 1, H_1 est la diagonale du Jacobien et H_2 est le Jacobien original :

$$H = r * H_1 + (1 - r) * H_2 \qquad (107)$$

Dans sa configuration actuelle, vingt pas sont utilisés pour faire passer r de 0 à 1. Lorsque $r = 0{,}05$, les valeurs propres de H sont liées aux variables d'état de la diagonale H_1 en minimisant la différence entre la valeur propre et l'élément de la diagonale associée. Pour tous les pas suivants, un estimé linéaire de la valeur de la prochaine valeur propre est construit à partir des deux derniers liens pour favoriser l'association au pas courant. Ainsi, sur un modèle relativement simple comme le modèle ASM1 – Décanteur ponctuel, les valeurs propres sont liées aux variables d'état comme indiqué par la Figure 18.

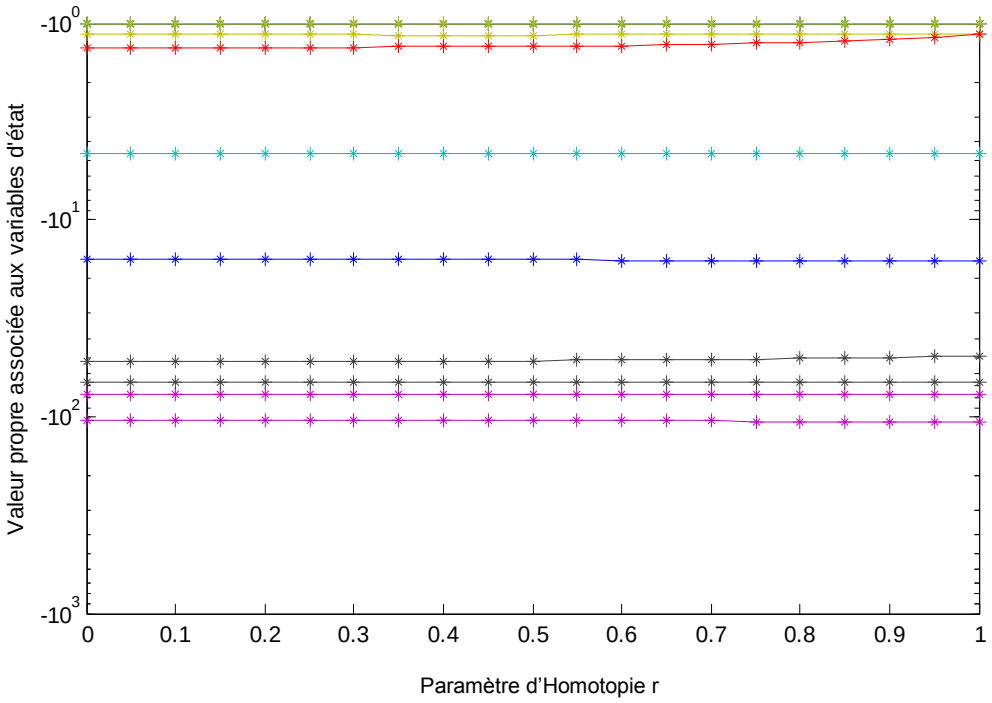


Figure 18 : Évolution des 14 valeurs propres sur le modèle ASM1 – Décanteur ponctuel en fonction du paramètre d'Homotopie r

La Figure 18 montre que les valeurs propres sont très proches des éléments de la diagonale et que vingt valeurs intermédiaires sont peut-être superflues. Cependant, sur un modèle plus complexe comme le Benchmark Simulation Model 1 (BSM1) comportant 108 variables d'état, l'évolution des valeurs propres demande plus de soins tel que montré à la Figure 19. Bien que les liens semblent tous assez bon, il est possible de trouver des zones où l'algorithme n'est pas adéquat, comme le montre la Figure 20.

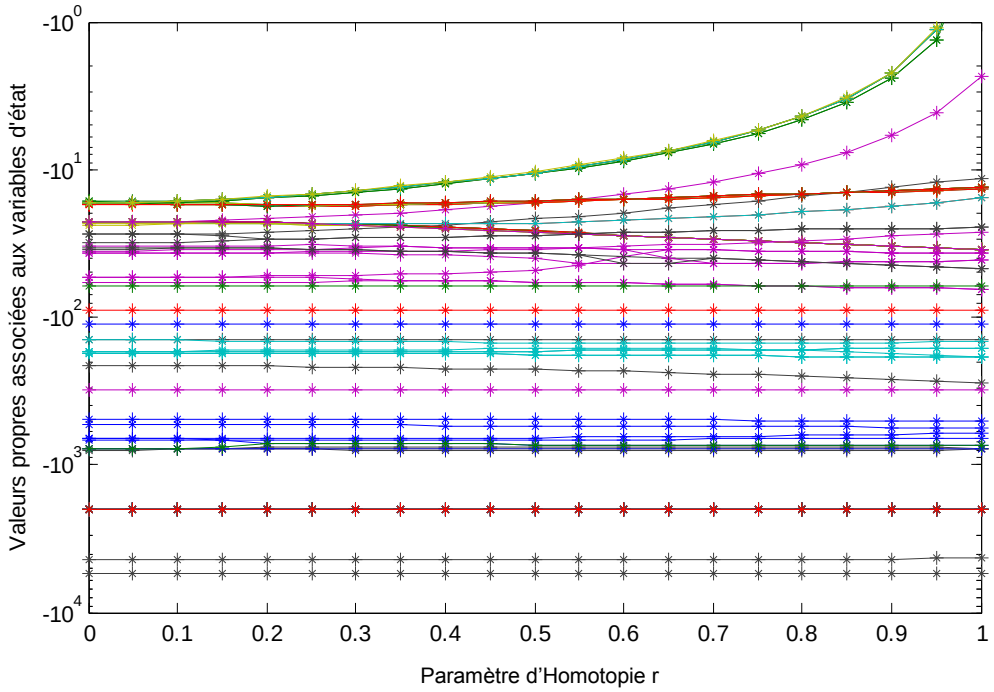


Figure 19 : Évolution des 108 valeurs propres sur le modèle BSM1 en fonction du paramètre d'Homotopie r

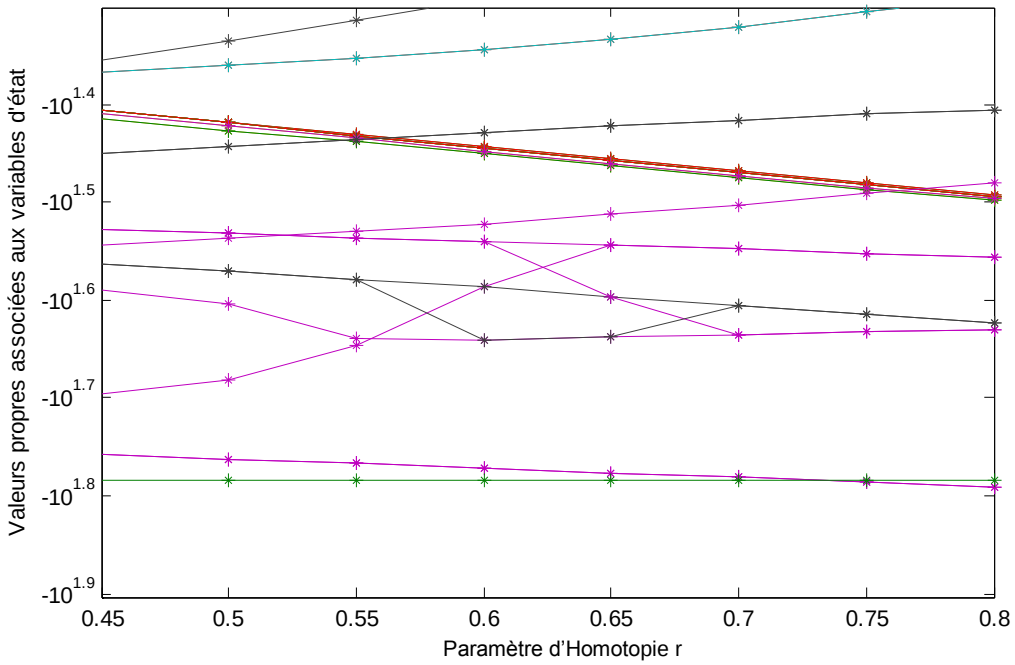


Figure 20 : Évolution des valeurs propres sur le modèle BSM1 en fonction du paramètre d'Homotopie r – Mauvaise association des valeurs propres aux variables d'état.

La Figure 20 montre quelques croisements qui n'auraient pas eu lieu si le nombre de pas en r avait été augmenté. Cependant, puisque le rôle de cette technique d'homotopie est de classer les variables d'état selon leurs valeurs propres associées, tant que les erreurs d'association ont lieues dans une même classe de vitesse, il n'y aura aucune erreur induite dans le reste de l'algorithme.

Lorsque les variables d'état sont associées à une valeur propre du Jacobien, il est possible de trier ces dernières et de diviser le modèle en deux sous-modèles dont le premier sera constitué des variables rapides et le second des variables lentes. Ce tri se base sur une valeur maximale des valeurs propres fournie par l'utilisateur de l'algorithme. Cette valeur est déterminée en considérant l'échelle de temps d'intérêt.

Puisque les valeurs propres sont définies comme étant l'inverse de la constante de temps τ d'une réaction, il est possible de déterminer une constante de temps représentant la limite entre les variables d'état dites lentes et les variables d'état rapides en fonction d'une échelle de temps d'intérêt. Ainsi, (Hesstvedt et al. 1978) propose de fixer la limite d'une constante de temps des variables rapides à 10% de l'échelle de temps d'intérêt. La simulation d'un modèle de station d'épuration sur une ou deux semaines requière habituellement une information sur les variables d'état à chaque demi-heure (environ 0,02 $jour$). Sur une base journalière, une valeur propre acceptable serait donc de l'ordre de 500 $jour^{-1}$, soit :

$$\frac{1}{0{,}1 * \tau} = \frac{1}{0{,}10 * 0{,}02} = 500 \qquad (108)$$

Les équations dites rapides seront alors résolues au régime permanent à chaque pas par l'algorithme DIRK. Pour se faire, le modèle passe d'un système d'ÉDO à un système d'ÉDA. La dérivée des variables d'état rapides est alors posée à zéro, ce qui transforme l'équation différentielle en une équation algébrique implicite.

Contrairement à la méthode proposée par (Steffens et al. 1997) qui analyse le Jacobien du modèle seulement au régime permanent ou à la méthode proposée par (Hesstvedt et al. 1978) qui suit l'évolution des constantes de temps des équations chimiques, l'analyse du modèle proposée ici est réalisée en continu et ne nécessite aucun accès aux constantes de temps. Ainsi, le Jacobien est calculé au point d'opération initial du modèle, soit en y_0, t_0 et

est analysé immédiatement en ce point. L'analyse sera ensuite recommencée à intervalles variables dépendant du temps où est rendu l'algorithme et du nombre de pas réalisé avec le modèle original ou réduit. De plus, des procédures doivent être mises en place pour confirmer que chaque réduction du modèle ne génère pas d'erreur trop importante dans la solution.

Cette méthode gagne donc en généralité puisque l'état du système peut changer radicalement tout en conservant un lien entre la valeur propre associée à une variable d'état et la façon la plus efficace de résoudre le modèle. Autrement dit, réévaluer la pertinence d'une réduction permet d'accélérer la solution d'un modèle dans le cas où certaines variables d'état développent une raideur durant l'évolution de la simulation. À l'inverse, si des variables perdent cette raideur, une erreur numérique significative peut apparaître si elles sont toujours résolues à titre de variable rapide. Et bien sûr, le pire des cas : certaines variables deviennent raide, ralentissant le calcul de la solution tandis que d'autres perdent leur raideur, diminuant la précision des résultats. Dans ces conditions, une procédure automatique de suivi du modèle montre des avantages certains.

4.1.3 Contrôle de la réduction automatique

Puisque le Jacobien représente une approximation locale du modèle, la réduction basée sur son étude est valide dans l'environnement immédiat du Jacobien. Dès que la solution s'en éloigne, il n'y a aucune garantie que la réduction est toujours valide. C'est pourquoi l'algorithme DIRK modifié pour réaliser la réduction automatique contient deux procédures distinctes pour éliminer au maximum les erreurs dues à de mauvaises réductions du modèle.

4.1.3.1 Évaluation régulière de l'évolution du Jacobien

Un modèle résolu en régime dynamique évolue en fonction de ses variables d'état. Pour cette raison, la réduction doit être réévaluée à intervalle régulier. Dans le cadre d'une simulation dynamique, la notion d'intervalle régulier prend un sens particulier. En effet, le temps de simulation n'est pas toujours proportionnel au temps de calcul. Dans le cadre d'une simulation de station d'épuration, des événements ponctuels comme une aération

intermittente ou une réaction en cuvée peuvent survenir à intervalles simulés réguliers. Puisque chacun de ces événements sont susceptibles de modifier la raideur des variables d'état, un contrôle de la raideur doit idéalement pouvoir réévaluer le modèle à chaque événement. L'algorithme de réduction du modèle est donc appelé minimalement à intervalle de temps simulé régulier. La notion de temps simulé faisant référence au temps que couvre une simulation.

Cependant, il arrive également que des conditions d'opération particulières viennent à ralentir l'évolution de la simulation en avançant à pas de temps trop petit. Ce ralentissement est généralement synonyme d'une raideur apparaissant dans le modèle. Pour éviter que l'algorithme DIRK ne perde trop de temps à solutionner un modèle raide, l'algorithme de réduction est également appelé à tous les n pas de temps complétés. Ainsi, si une raideur apparaît, elle sera détectée en au plus n pas et le modèle réduit pourra la traverser efficacement.

Le couplage de ces deux contrôles permet de garantir une évaluation régulière de l'évolution de la raideur du modèle.

4.1.3.2 Changement important dans les valeurs propres suite à la réduction du modèle

Le premier problème courant tire son origine dans la nature très locale du Jacobien. Ainsi sous certaines conditions, des variables d'état peuvent présenter une dynamique locale très rapide alors que cette dynamique est typiquement assez lente. Dans ces conditions, le processus de réduction automatique du modèle considérera ces variables d'état comme étant rapides et elles seront résolues à l'équilibre. Une telle réduction hâtive peut mener à des comportements comme celui présenté à la Figure 21.

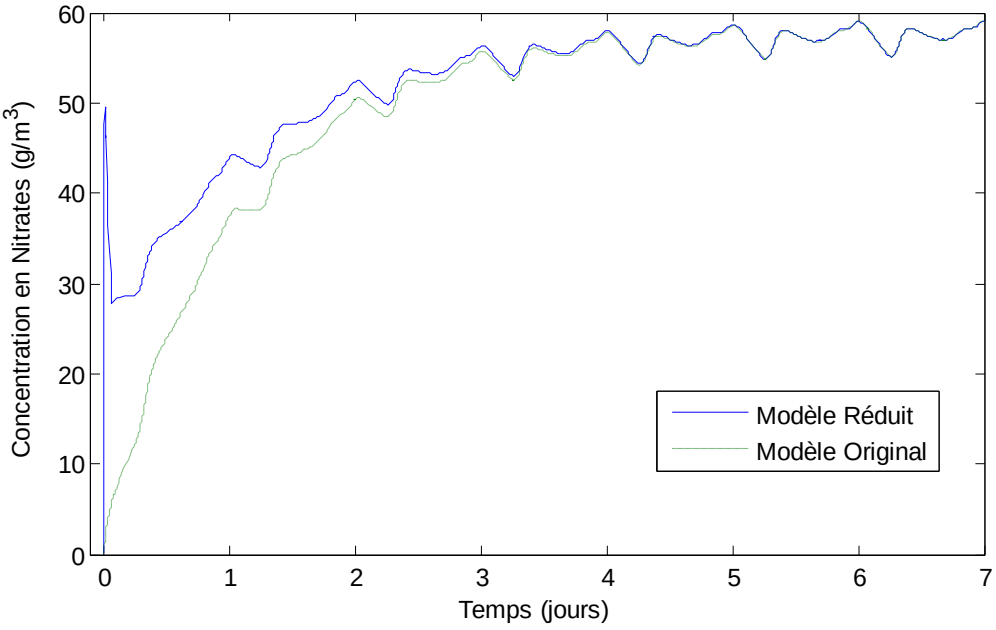


Figure 21 : Courbe de la concentration des nitrates simulée par un modèle réduit et un modèle original

La Figure 21 montre la solution d'une variable d'état qui est initialement jugée rapide. Cependant, cet état est dicté par les conditions locales du modèle plutôt que par la vitesse intrinsèque de la variable d'état. Ainsi, après un seul pas, sa valeur propre est grandement relaxée et n'appartient plus à la catégorie des variables d'état rapides.

Comme l'indique la Figure 21, l'erreur induite par la réduction du modèle est trop importante si l'échelle de temps d'intérêt est de l'ordre de l'heure.

Un tel cas est détecté automatiquement par l'algorithme en recalculant un Jacobien à jour après que l'algorithme d'intégration ait calculé le pas suivant. Les valeurs propres du Jacobien sont recalculées et le nombre de valeurs propres élevées (correspondant aux variables d'état rapides) est comparé au nombre de variables résolues au régime permanent. Si ce nombre a diminué, une ou plusieurs variables d'état ont vu leur constante de temps augmenter et la réduction du modèle n'est plus valide. L'algorithme rejette donc le dernier pas calculé et recommence avec le modèle original. Un cas comme celui de la Figure 21 n'est donc pas accepté par l'algorithme. Par la suite, l'état du modèle est réévalué selon les balises données à la section précédente (section 4.1.3.1). Le fait de résoudre le modèle

durant un court temps ou un faible nombre de pas permet généralement de passer par-dessus la difficulté passagère à un coût de calcul acceptable.

4.1.3.3 Contrôle de la solution en fonction du signe des variables d'état

La constante de temps d'une variable d'état changeant très rapidement n'est pas le seul cas menant à une dégradation des résultats de la simulation d'un modèle réduit. En effet, contrairement à un processus d'intégration où l'erreur sur le pas suivant est simple à contrôler, la recherche de la position d'équilibre d'une ou plusieurs variables d'état ne peut garantir que la solution trouvée soit la bonne. Ainsi, sur un modèle ASM2d utilisant une aération intermittente, donc une concentration en oxygène dissous variant entre 0 et 6 g/m3, la concentration d'oxygène a été calculée comme à la Figure 22.

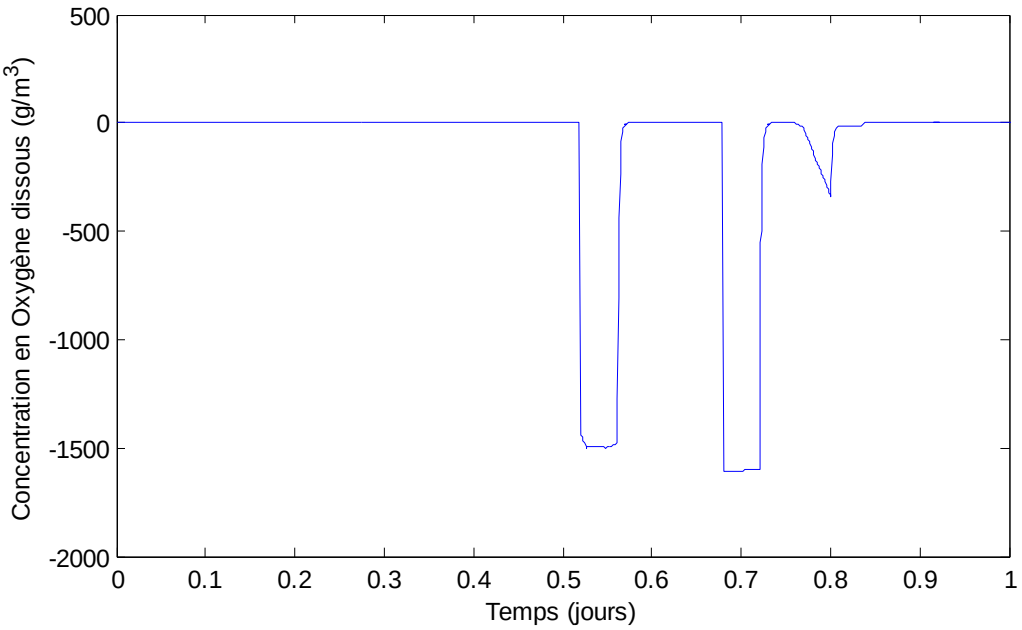


Figure 22 : Concentration en oxygène dissous dans un modèle ASM2d avec la réduction de modèle automatique

La Figure 22 montre une solution acceptable d'un point de vue mathématique, mais inacceptable du point de vue de la modélisation de la réalité puisqu'une concentration négative apparaît. La solution proposée et implémentée dans l'algorithme revient à refuser à un modèle réduit tout changement de signe dans la solution des variables d'état. Ce critère se base sur le fait que l'algorithme a été écrit dans un but général, mais axé sur la

simulation de modèles biologiques. Puisque les variables d'état de ce type de modèle sont généralement de masses ou des concentrations, un changement de signe n'est généralement pas acceptable. Cependant, comme des exceptions peuvent exister (variables d'état représentant des contrôleurs, par exemple), il est nécessaire de permettre la simulation d'un modèle réduit comportant des variables d'état négatives. Ainsi le test utilisé est décrit à l'équation (109) :

$$y_n * y_{n-1} \geq 0 \qquad (109)$$

Ici, n correspond au pas où est rendue la simulation. Si un changement de signe apparaît, le pas est rejeté et le modèle original est utilisé pour continuer la simulation jusqu'à l'évaluation du modèle suivante (section 4.1.3.1). Si les conditions le dictent, une variable d'état traversera donc zéro avec le modèle original et une nouvelle réduction pourra être exécutée en conservant ce nouveau signe.

Dans le cas de l'oxygène dissous de la Figure 22, ce contrôle permet de confirmer que la concentration en oxygène demeure positive, comme en fait foi la Figure 23.

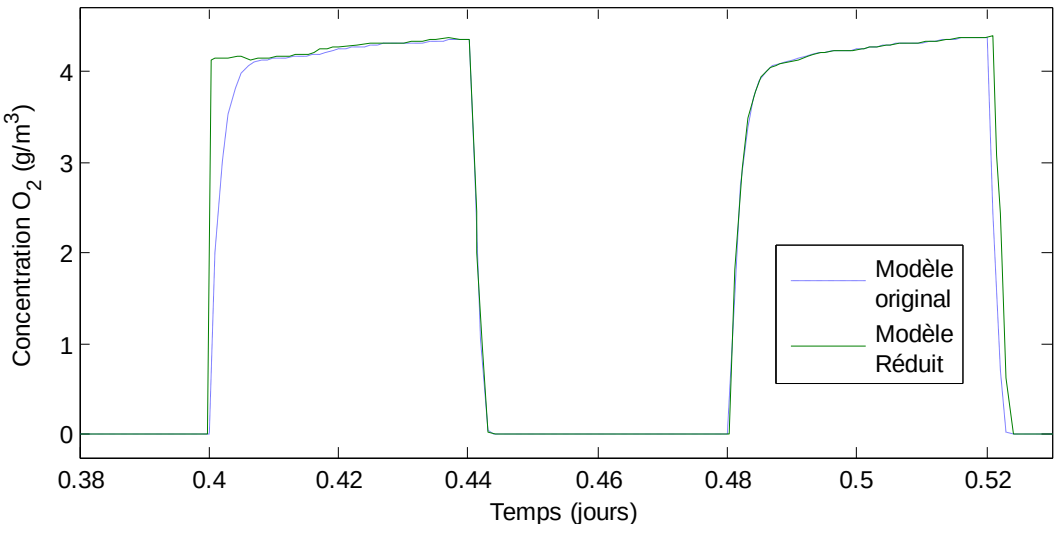


Figure 23 : Concentration en oxygène dissous simulée avec un modèle réduit et le modèle original

Sur la Figure 23, le premier pic d'oxygène a été calculé à l'aide du modèle réduit. En arrivant au second, par contre, un problème numérique est apparu et le modèle original a été utilisé pour passer par-dessus cette difficulté.

4.1.4 Performances de l'algorithme

Les performances d'un algorithme dépendent énormément du modèle sur lequel cet algorithme est appliqué. De plus, plusieurs critères peuvent être utilisés pour comparer deux algorithmes. Cette section en retient trois :

1. Le temps de calcul nécessaire pour simuler un modèle,
2. Le nombre de fois que le modèle doit être évalué et
3. Le nombre de pas réalisé par le modèle.

Le premier critère paraît à première vue le plus important. Cependant, il dépend énormément de l'ordinateur sur lequel la simulation est réalisée et également de l'état de l'ordinateur (nombre de logiciels ouverts en parallèle, etc.). Le second critère est plus objectif puisqu'il ne dépend pas de l'ordinateur. Le nombre d'évaluations du modèle est donc un critère typique pour évaluer la performance d'un algorithme. Finalement, le nombre de pas réalisé donne un aperçu de la taille moyenne de chaque pas que l'algorithme calcule. Dans le cadre d'un algorithme expérimental tel que l'algorithme DIRK et la procédure de réduction, le nombre de pas donne plutôt une indication des performances qui pourraient être atteintes une fois toutes les optimisations algorithmiques possibles réalisées.

Finalement, des études ont été réalisées pour comparer différents algorithmes (voir (Claeys 2008b), par exemple). Dans le cas présent, la version originale de l'algorithme DIRK sera comparée à la réduction automatique sur l'algorithme DIRK et sera comparée à un algorithme performant de référence, l'algorithme CVODE.

Les trois algorithmes sont testés sur deux modèles, soit le modèle ASU et le Benchmark Simulation Model 1 (BSM1). Le modèle BSM1 est intéressant de par sa taille (108 variables d'état) qui correspond aux petits modèles représentant des usines réelles. Le second modèle utilise moins de variables d'état, avec 30 variables, mais introduit un contrôleur (aération intermittente).

4.1.4.1 Modèle ASU

Le modèle ASU utilise une tolérance relative de 10^{-4} sur les algorithmes DIRK avec et sans réduction automatique du modèle et 10^{-9} sur les tolérances absolue et relative de

l'algorithme CVODE pour minimiser la déviation observée sur cet algorithme. Le modèle réduit considère une variable d'état comme étant rapide si sa valeur propre associée est supérieure à 1000 $jour^{-1}$, soit une constant de temps d'environ une minute.

Le premier résultat de cette simulation (Tableau 7) est que l'algorithme DIRK est au moins cinq fois plus lent que l'algorithme de référence CVODE. Cet ordre de grandeur se retrouve également dans le nombre d'appel du modèle pour chaque algorithme. Ainsi, dans le cadre de ce modèle précis, l'intérêt d'utiliser l'algorithme DIRK est très limité. Cependant, lorsque des simulations à faible précision sont acceptées, l'utilisation de l'algorithme DIRK peut offrir une meilleure stabilité de la solution à un coût de calcul comparable à l'algorithme CVODE, comme le montre le Tableau 8. Ainsi, à faible tolérance, l'algorithme DIRK offre des performances comparables à l'algorithme CVODE.

Tableau 7 : Résultats sur le modèle ASU pour l'algorithme DIRK avec et sans réduction et pour l'algorithme de référence CVODE

	DIRK	DIRK + Réduction	CVODE
Temps de simulation (s)	46	109	9
Nombre d'évaluation du modèle	1 961 434	3 595 127	381 043
Nombre de pas calculé	101 244	59 519	225 000

Tableau 8 : Résultats sur le modèle ASU avec l'algorithme DIRK à deux tolérances différentes

	DIRK Tolérance = 10^{-4}	DIRK Tolérance = 10^{-3}
Temps de simulation (s)	46	12
Nombre d'évaluation du modèle	1 961 434	447 842

Nombre de pas calculé	101 244	27 324

La réduction automatique de modèle mène donc à une dégradation importante des temps de calculs. En effet, dans sa version originale, l'algorithme DIRK nécessite en moyenne 19,4 évaluations du modèle par pas tandis que la réduction automatique en requière en moyenne 60,9. Ainsi, en ne considérant que le nombre d'évaluation du modèle, pour que la réduction devienne intéressante et présente un intérêt, le nombre de pas doit être au moins six fois plus faible sur le modèle réduit que sur le modèle original.

Dans son implémentation actuelle, la réduction du modèle requière des évaluations supplémentaires d'abord pour contrôler la solution lorsque les valeurs propres sont évaluées avant et après la réduction puisque cette technique nécessite l'évaluation numérique du Jacobien, donc de 30+1 évaluations supplémentaires (n variables d'état plus 1). De plus, le modèle ASU réduit automatiquement voit entre une et trois variables d'état (les matières fermentables `S_F`, l'oxygène dissous `S_O` et l'acétate `S_A`) être déclarées rapides selon le point d'opération.

Bien que l'algorithme DIRK soit capable de solutionner des systèmes d'ÉDO et d'ÉDA, le second système pose une difficulté supplémentaire. En effet, une méthode de Runge-Kutta construit la solution à un pas en calculant la somme pondérée des pentes en différents points (équation (28)). Or, les variables algébriques ont une pente nulle par définition $\left(\frac{dy}{dt} = 0\right)$. Il est donc impossible de calculer leur valeur finale sur la base d'une somme pondérée des pentes. Le mieux que l'algorithme DIRK soit en mesure de faire est d'offrir d'excellents estimés initiaux. Néanmoins, il faut généralement deux à quatre itérations de Newton-Raphson pour trouver la solution finale des équations algébriques. Ces itérations nécessitent l'évaluation du Jacobien des variables concernées (entre une et trois), donc de quatre à seize évaluations supplémentaires du modèle par pas.

4.1.4.2 Modèle BSM1

Le modèle BSM1 est calculé en imposant une tolérance relative de 10^{-4} sur les algorithmes DIRK avec et sans réduction de modèle. Lorsque la réduction de modèle est

permise, la limite entre les variables rapides et lentes se situe à 1000 jours-1, soit équivalant à une constante de temps d'environ 90 secondes. Finalement, l'algorithme CVODE se voit imposé une tolérance absolue et relative de 10^{-9} pour éviter toute divergence des résultats.

Tableau 9 : Résultats sur le modèle BSM1 pour l'algorithme DIRK avec et sans réduction et pour l'algorithme de référence CVODE

	DIRK	DIRK + Réduction	CVODE
Temps de simulation (s)	44	193	14
Nombre d'évaluation du modèle	956 972	6 639 413	498 761
Nombre de pas calculé	94 443	90 113	246 712

Le Tableau 9 montre que dans sa version actuelle, l'algorithme DIRK prend environ trois fois plus de temps à calculer la solution que l'algorithme de référence CVODE. Cependant, le fait que le temps de simulation soit trois fois supérieur alors que le nombre d'évaluation du modèle ne dépasse pas le double du nombre d'évaluations de modèle réalisé par CVODE montre que l'algorithme DIRK doit fournir beaucoup plus d'effort interne par évaluation de modèle. Ce résultat s'explique par le fait que l'algorithme DIRK doit résoudre jusqu'à quatre fois l'ensemble d'équations algébriques implicites de l'équation (29). Ainsi, les opérations matricielles (calcul du Jacobien, décomposition LU, etc.) réalisées par l'algorithme sont très importantes.

La réduction automatique du modèle montre cependant une dégradation significative des résultats. En effet, les temps de calcul sont plus de dix fois plus importants que ceux

nécessaire à l'algorithme de référence CVODE. De plus, le nombre de calcul du modèle très imposant montre qu'une optimisation de l'algorithme est nécessaire. Ces appels supplémentaires sont principalement dus à l'évaluation numérique du Jacobien qui requiert 109 évaluations du modèle chaque fois. De plus, le modèle réduit résout entre 30 et 33 équations d'état à l'équilibre à chaque pas. Or ces équations nécessitent l'utilisation d'un solveur interne pour la solution finale. Sauf que contrairement au modèle ASU où seules une à trois équations algébriques devaient être résolues, devoir trouver la solution simultanée à 30 variables exige la génération d'un Jacobien beaucoup plus imposant. Si deux seules itérations sont nécessaires pour atteindre l'équilibre, 60 évaluations du modèle sont requises uniquement pour les besoins du Jacobien à chaque pas, ce qui couvre l'essentiel de la différence entre le nombre d'évaluation par pas pour le modèle original (10 évaluations par pas pour le modèle original contre 73 pour le modèle réduit).

Le nombre de pas reste du même ordre de grandeur entre la solution du modèle complet et du modèle réduit. En supposant que ces pas aient une grandeur moyenne constante, le pas moyen couvre environ 13 secondes, soit une valeur du même ordre que l'échelle de temps d'intérêt de 90 secondes. De tels résultats montrent que la réduction automatique du modèle a très peu de potentiel sur des modèles simples, c'est-à-dire sans éléments susceptibles de dégrader les performances de l'algorithme de façon significative (variables très raides comme le calcul du pH ou des conditions de sorption/désorption particulières, bruit dans les entrées, contrôleurs particuliers, etc.).

4.1.5 Discussion sur l'algorithme DIRK et sur la réduction automatique d'un modèle

Dans sa forme actuelle, la réduction automatique d'un modèle ne peut être appliquée à tout modèle. En effet, les calculs supplémentaires requis par l'algorithme sont très lourds. Cependant, sur des modèles assez raides un modèle réduit pourrait démontrer son intérêt. Les résultats obtenus sur le modèle ASU montrent que s'il est possible d'avancer à l'aide de pas de temps six fois plus importants ou plus, le modèle réduit sera avantagé par rapport au modèle original.

L'algorithme DIRK doit également justifier son intérêt par rapport à d'autres algorithmes performants comme l'algorithme CVODE. Dans ce cas, l'algorithme DIRK n'est pas approprié lorsque l'erreur ne peut être tolérée puisque le coût pour forcer les tolérances est très important. Cependant, à tolérance moindre, l'algorithme DIRK montre d'excellentes performances, nécessitant des temps de calcul de l'ordre de l'algorithme CVODE sans, pour autant, afficher une dérive des variables d'état pouvant atteindre les 15%. C'est pourquoi il est recommandé d'utiliser l'algorithme DIRK lorsque des simulations à tolérance moindre sont acceptables.

En conclusion, il faut noter que les travaux sur l'algorithme DIRK et sur la réduction automatique d'un modèle sont des travaux exploratoires. Ainsi, bien que très avancés, ces travaux ne sont pas complétés. Des percées majeures peuvent donc encore être espérées. Notamment, s'il est possible de développer des techniques de calcul du Jacobien symbolique (voir section 4.3). De plus, la décomposition LU est à la base de toutes les opérations matricielles sur l'algorithme DIRK. La décomposition QR n'a pas été implémentée, mais devrait l'être pour pouvoir profiter de sa capacité à mettre à jour une décomposition sans avoir à recalculer le Jacobien en entier.

4.2 Étude du Régime Permanent

4.2.1 Régimes permanents des modèles à l'étude

Des trois modèles étudiés en régime permanent (Prédateur – Proie, Acide – Base, ASM1), un seul peut être résolu sans problème par les algorithmes disponibles sur la plateforme Tornado (algorithme de Broyden et algorithme Hybride), soit le modèle Prédateur – Proie. Les modèles ASM1 et Acide – Base présentent un Jacobien singulier. L'algorithme de Broyden décrit dans (Press et al. 1994) renvoie un code d'erreur dès que la singularité est détectée tandis que l'algorithme Hybride tente de calculer une solution malgré la singularité. Néanmoins, chaque modèle présente plus d'une solution au régime permanent possible du point de vue mathématique. Ces résultats sont présentés aux Tableau 10, Tableau 11 et Tableau 12.

Tableau 10 : Régimes permanents du modèle Prédateur - Proie

Population de Proie (nb. d'individu)	Population de Prédateur (nb. d'individu)
0	0
110 000	0
48095	619

Tableau 11 : Régimes permanents du modèle Acide - Base

H^+ (mol/L)	OH^- (mol/L)	NH_3 (mol/L)
1.27×10^{-7}	7.83×10^{-8}	4.92×10^{-8}
-1.27×10^{-7}	-7.83×10^{-8}	-4.92×10^{-8}
-3.87×10^{-10}	-2.58×10^{-5}	2.58×10^{-5}

Tableau 12 : Quelques régimes permanents du modèle ASM1 et Décanteur ponctuel

S_ALK (g)	S_ND (g)	S_NH (g)	S_NO (g)	S_O (g)	S_S (g)	X_BA (g)	X_BH (g)	X_ND (g)	X_P (g)	X_S (g)
9 593	824,9	49,53	5 741	7 001	9 197	10 147	67 003	43,22	315	471
9 231	-28,30	323,3	11 084	7 531	115 481	11 332	-5 028	1 865	848	18 204
3 789	-18,15	-38132	48 808	4 330	115 592	20 286	-5 053	1 864	854	18 219
9 280	2 606	278,8	10 353	7 591	133 351	11 159	-4 861	4,614	853	45,07
3 739	2 621	-38 882	48 763	4 332	133 477	20 275	-4 885	4,66	859	45,66
7 565	2 623	-11 180	22 903	962,2	11 091	17 142	379 735	-48 060	13 165	-539 743
9 350	2 674	3 457	12 550	1 187	11 057	14 474	429 961	-55 705	14 770	-626 199

Si le modèle Prédateur – Proie possède trois régimes permanents ayant une signification physique, il en va autrement des deux autres modèles. Les modèles Acide – Base et ASM1 ne possèdent qu'une seule solution avec une signification physique. En effet, une concentration négative ne correspond à aucune réalité physique. La seule solution acceptable pour ces deux modèles est donc la première présentée aux Tableau 11 et Tableau 12. Les autres solutions sont des artéfacts mathématiques qui résolvent le système d'équation, mais qui n'offrent aucune information utilisable.

Les valeurs d'équilibre indiquées au Tableau 12 ont été calculées en utilisant les estimés initiaux de base (voir Matériel et Méthode, section 0) et en faisant varier les estimés initiaux des concentrations de l'oxygène dissous (S_O) entre $10^{-3}\ g/m^3$ et $10\ g/m^3$ et de la matière organique soluble (S_S) entre $10^{-3}\ g/m^3$ et $100\ g/m^3$. Bien que ces estimés initiaux soient des valeurs acceptables, tous les algorithmes montreront de meilleurs taux de convergence lorsque les estimés initiaux sont proches des solutions. De plus, les concentrations en matière inerte soluble (S_I), en matière inerte particulaire (X_I) et le volume d'eau ne sont pas affichés au Tableau 12 puisqu'ils sont constants peu importe l'équilibre atteint.

Ces différentes solutions sont obtenues en faisant varier les estimés initiaux donnés à l'algorithme Hybride de MINPACK. Il est clair qu'un bon algorithme devra permettre de converger vers la solution d'intérêt. C'est dans ce but que quelques modifications automatiques du modèle ont été proposées.

4.2.2 Résultats sur les algorithmes disponibles et modifications apportées

Tel que vu dans la revue de littérature, les algorithmes de Broyden et Hybride sont très similaires. Ils utilisent tous deux les dérivées des équations différentielles nécessaires à la construction de la matrice Jacobienne, ils décomposent cette dernière par la méthode dite QR et ils cherchent à minimiser les résidus des équations différentielles. Cependant, l'algorithme de Broyden utilise la technique de la pente descendante maximale alors que l'algorithme Hybride de la bibliothèque MINPACK utilise plutôt la technique de la région de confiance couplée à la technique du zigzag (dog-leg, en anglais). Pour les modèles simples, l'algorithme de Broyden offre des performances similaires à l'algorithme Hybride. Cependant, lorsque le modèle est très non-linéaire, la technique de la zone de confiance permet de contrôler la progression de la solution en fonction de la non-linéarité du modèle.

Les deux algorithmes dans leur version originale sont donc testés sur les trois modèles. La sensibilité des algorithmes aux estimés initiaux fait en sorte que plusieurs tentatives se soldent par un échec de l'algorithme. Cet échec peut survenir pour de nombreuses raisons. Les plus courantes sont une variable d'état prenant une valeur indéfinie (NaN « Not a Number », INF « Infini ») résultant d'une division par zéro, par exemple. Ou encore, simplement une convergence trop lente aux yeux de l'algorithme qui estime alors qu'aucune solution ne peut être calculée. Dans tous les cas, la plateforme Tornado ne renvoie qu'un code d'erreur. La seule information fournie est donc un échec de l'algorithme. Dans les figures qui suivent, les résultats d'erreur font référence à ce type d'échec.

4.2.2.1 Résultats des algorithmes sur le modèle Prédateur – Proie

Les algorithmes sont évalués en faisant varier les estimés initiaux des populations de prédateurs et de proies entre 0 et 100 000 pour les proies et entre 0 et 1000 pour les prédateurs avec vingt valeurs espacées également dans ces intervalles, pour un total de 400 évaluations. Ces calculs ont été réalisés sur la plateforme Tornado à l'aide de l'expérience virtuelle de type `ExpScenSSRoot`. Le détail de l'expérience virtuelle, en format XML se retrouve à l'Annexe B. Dans ces conditions, le nombre d'essai convergeant à une solution

est comptabilisé de même que le nombre moyen d'évaluations du modèle nécessaire pour y converger.

Tableau 13 : Performances des algorithmes Broyden et Hybride sur le modèle Prédateur - Proie

Algorithme	$\frac{Nombre\ de\ convergences}{Nombre\ d'essais}$	Nombre moyen d'évaluations du modèle
Broyden	400/400	22,35
Hybride	400/400	19,76

Ainsi, sur un modèle simple comme le modèle Prédateur – Proie, les deux algorithmes convergent sans problème peu importe l'algorithme utilisé. L'algorithme de Broyden est légèrement plus lent, mais la nuance reste très faible.

Il est également possible de visualiser les solutions où ont convergés les algorithmes en fonction des estimés initiaux. Les Figure 24 et Figure 25 montrent les tendances générales de convergence en fonction des estimés initiaux.

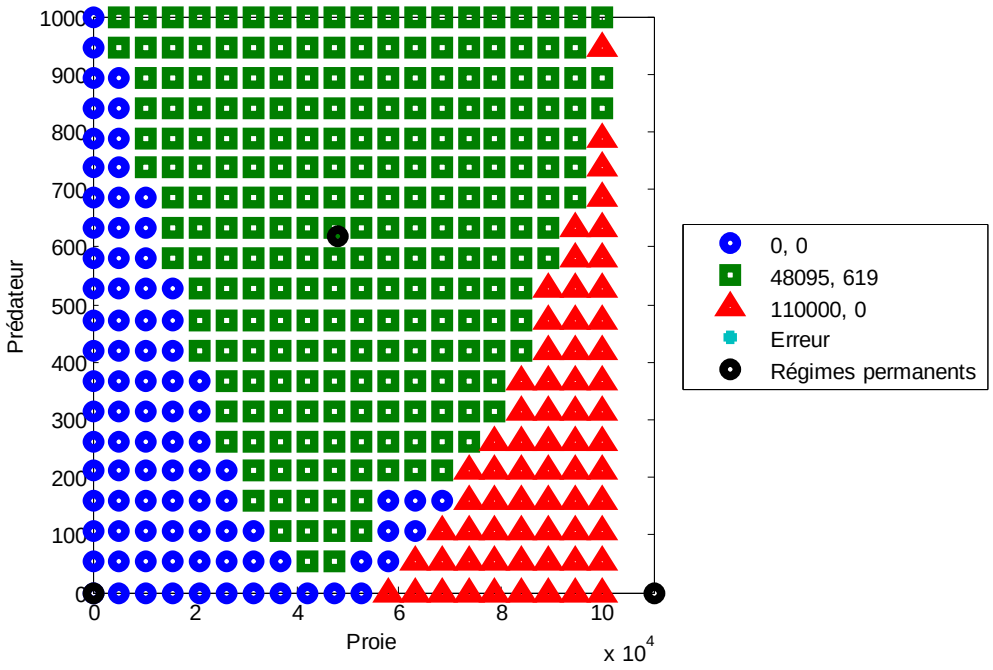


Figure 24 : Algorithme de Broyden et convergence vers les différentes solutions, modèle Prédateur – Proie

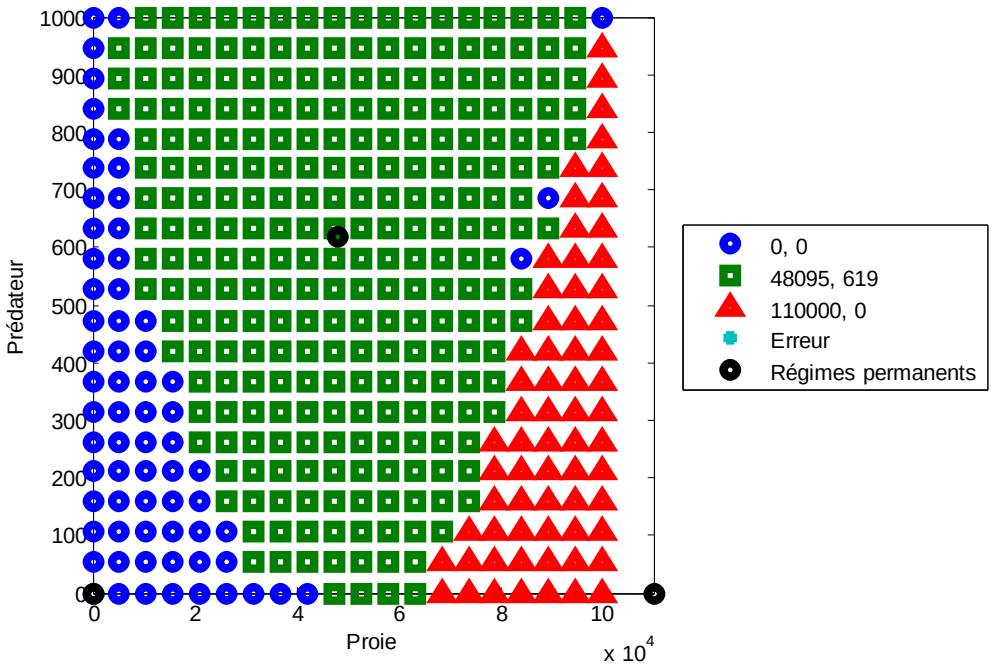


Figure 25 : Algorithme Hybride et convergence vers les différentes solutions, modèle Prédateur – Proie

L'étude de ces figures montre un comportement similaire entre les algorithmes de Broyden et Hybride. Les deux figures présentent des zones de convergence distinctes pour chaque solution. Il est donc possible d'observer que la solution trouvée par l'algorithme dépend très fortement des valeurs des estimés initiaux donnés à l'algorithme.

4.2.2.2 Résultat des algorithmes sur le modèle Acide – Base

Pour travailler avec le modèle Acide – Base, il faut d'abord renforcer considérablement les seuils de tolérance. En effet, la tolérance par défaut des algorithmes est de l'ordre de 10^{-6}. De plus, le critère d'arrêt de l'algorithme Hybride est construit à l'aide de l'équation (110).

$$\frac{\sqrt{\sum_i \left(\max[1, y_i + f_i] * (-f_i)\right)^2}}{0{,}5 * \sqrt{\sum_i y_i^2}} \leq Tolerance \qquad (110)$$

Compte tenu de la division de l'erreur par la norme euclidienne des variables d'état, un modèle présentant des variables d'état d'amplitude très variées comme le modèle Acide –

Base nécessite l'usage d'une tolérance très restrictive. C'est pourquoi la tolérance est posée à 10^{-13}.

Puisque le modèle acide – base contient trois variables d'état, pour des raisons de lisibilité, seules les estimés initiaux des concentrations des ions H^+ et OH^- sont variés. L'estimé initial de la concentration de la molécule NH_3 est gardé constant à 5×10^{-5}. Cette valeur correspond à la moitié de la concentration de l'azote totale. Il s'agit donc d'un estimé initial crédible en considérant un manque d'information supplémentaire.

La formulation du modèle Acide –Base retourne un message d'erreur lorsque l'algorithme de Broyden tente de le résoudre à cause de l'équation de contrainte. En effet, la contrainte sur la charge, donnée par l'équation (99), est une équation algébrique implicite.

$$0 = H^+ + NH_4^+ - OH^- + Z^+ \qquad (111)$$

Puisque les algorithmes de Broyden et Hybride recherchent les valeurs des variables d'état pour lesquelles les équations différentielles sont toutes nulles, l'équation (99) est écrite sous la forme de l'équation différentielle d'une variable fictive. L'algorithme de Broyden ne parvient pas à résoudre ce modèle parce que toutes les dérivées partielles par rapport à cette variable fictive sont nulles, ce qui génère un Jacobien singulier. Et puisqu'une matrice singulière est synonyme de système sous-déterminé, l'algorithme de Broyden estime qu'aucune solution satisfaisante ne peut être calculée. Dans les faits, la seule variable pouvant être indéterminée est cette variable fictive ajoutée. L'indétermination n'est donc aucunement une contrainte à la solution du modèle.

L'algorithme Hybride est plus flexible dans son implémentation et permet de solutionner un tel système et retourne les résultats affichés à la Figure 26.

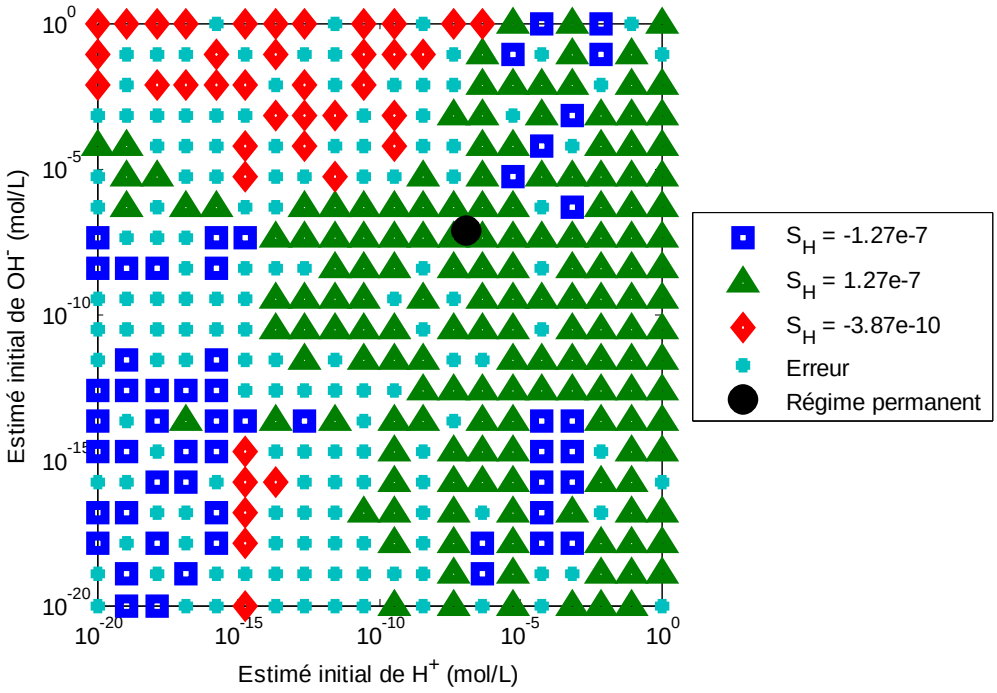


Figure 26 : Algorithme Hybride et convergence vers les différentes solutions, modèle Acide – Base

Il est possible de tirer quelques conclusions de la Figure 26. Tout d'abord, les estimés initiaux convergeant vers la solution physique ($S_H = 1.27 \times 10^{-7}$) semblent assez uniformément distribués bien que plus présents autours de sa solution finale. La solution inverse ($S_H = -1.27 \times 10^{-7}$) est présente, mais elle est très peu uniforme. Une hypothèse pour justifier sa présence de manière aussi disparate est qu'une correction trop forte appliquée à l'estimé par l'algorithme fasse changer la solution de zone de convergence.

Finalement la troisième solution ($S_H = -3.87 \times 10^{-10}$) semble posséder une zone de convergence assez bien définie aux estimés initiaux très faibles de S_H et très élevés de S_OH.

Le grand nombre d'expérience avortée est principalement dû à la variable fictive ajoutée au modèle. En effet, puisque cette variable n'apparaît nulle part dans le modèle, l'algorithme Hybride lui donne des valeurs quasi aléatoires qui varient entre $-1{,}84 \times 10^{26}$ à $8{,}46 \times 10^{25}$. Bien que ces valeurs n'affectent pas le reste du modèle, l'impossibilité de leur fixer une valeur oblige régulièrement l'algorithme Hybride à déclarer forfait.

4.2.2.3 Résultat des algorithmes sur le modèle ASM1

Le modèle ASM1 présente une complexité telle qu'il est très difficile de présenter les résultats du régime permanent lorsque toutes les variables d'état sont loin de leur valeur finale. En effet, si les quatorze variables d'état devaient prendre deux valeurs différentes comme estimé initial, pour tester toutes les combinaisons possibles, il faudrait 16384 essais distincts (2^{14}). Puisque le modèle ASM1 est généralement bien connu de ses utilisateurs, il a été décidé de ne varier les estimés initiaux que par paires pour améliorer la lisibilité. Les autres variables d'état sont gardées à des valeurs proches, mais approximatives, de la valeur du régime permanent. Ces valeurs sont affichées au Tableau 14.

Tableau 14 : Variables d'état du modèle ASM1 et décanteur ponctuel. Valeurs au régime permanent et estimé initial fourni aux algorithmes de calcul du régime permanent.

Variable d'état	Valeur au Régime permanent (g)	Valeur au Régime permanent $\left(\frac{g}{m^3}\right)$	Estimé initial (g)
H2O	1e+009		1e+009
S_ALK	9593,6	9,593	9000
S_I	10000	10	10000
S_ND	824,923	0,824	800
S_NH	49,5313	0,050	50
S_NO	5741,04	5,741	5000
S_O	7001,52	7,002	7000
S_S	9197,03	9,197	9000
X_BA	10147,7	10,148	10000
X_BH	67003,5	67,004	67000
X_I	10000	10	10000
X_ND	43,2298	0,0432	40
X_P	3152,23	3,152	3000
X_S	471,74	0,472	400

Encore une fois, l'idée générale est de donner des valeurs raisonnables. Que ces valeurs soient si proches de la valeur finale n'est pas un problème en soit comme le démontreront les résultats qui suivent puisqu'en variant un ou deux estimés initiaux, il est possible de faire diverger la solution finale de façon spectaculaire. De plus, trois variables d'état sont déjà à leur valeur finale : la masse d'eau (H20) et les composés inertes solubles (S_I) et

non-solubles (X_I). Ces choix sont logiques puisque le modèle ASM1 ne génère ni ne consomme aucun de ces composés. Leur fonction d'état est donc un bilan de masse sur les entrées et les sorties. Il est logique qu'au régime permanent, les entrées égalent les sorties.

Le Jacobien du modèle calculé par l'algorithme de Broyden est déclaré singulier à cause du volume fixe du modèle. Ainsi, l'équation différentielle de la masse d'eau présente dans le réacteur est donnée par

$$\frac{dH_2O}{dt} = Q_{in} - Q_{out} \tag{112}$$

Et puisqu'un volume fixe implique que le débit de sortie soit égal au débit d'entrée, l'équation (112) est nulle en tout temps, de même que toutes ses dérivées partielles présentes dans le Jacobien.

Ce fait empêche l'utilisation de l'algorithme de Broyden pour calculer le régime permanent du modèle. Les résultats seront donc le fait exclusif de l'algorithme Hybride. De plus, tel qu'indiqué à la section 4.2, l'abondance de solutions possibles différentes ne permet pas de toutes les afficher. C'est pourquoi une fonction objectif (équation (113)) est construite pour estimer la distance entre la solution calculée et la solution réelle.

$$FonctionObjectif = \sum_{i=1}^{14} \frac{|y_i - y_{iRP}|}{y_{iRP}} \tag{113}$$

Les résultats sont donc triés en trois catégories : ceux convergeant à la solution ayant un sens physique indiquée au Tableau 14, ceux convergeant à une autre solution affichée au Tableau 12 et les expériences qui retournent un code d'erreur de la part de l'algorithme. Pour les mêmes raisons qu'avec le modèle Acide – Base, la tolérance donnée à l'algorithme Hybride doit être très serrée pour offrir une convergence acceptable. En effet, tel qu'indiqué au Tableau 14, il n'y a pas moins de huit ordres de grandeurs entre la plus grande variable d'état et la plus petite au régime permanent. Ainsi, une tolérance de 10^{-13} permet d'espérer que l'algorithme ne s'arrête pas avant d'atteindre le régime permanent.

Pour illustrer les performances de l'algorithme Hybride, les estimés initiaux des variables d'état X_BH et X_S sont variées sur la plage 10^4 - 5×10^6 pour X_BH et 10^2 - 10^6 pour X_S selon une échelle logarithmique.

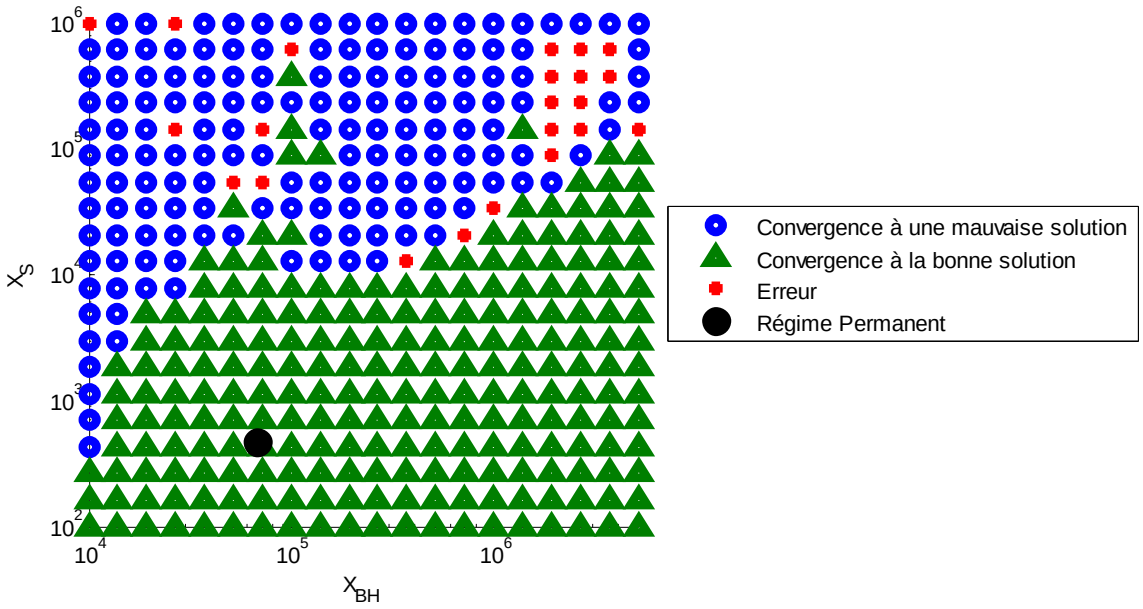


Figure 27 : Algorithme Hybride et convergence vers la solution souhaitée ou non, modèle ASM1 et décanteur ponctuel.

Encore une fois, la zone de convergence est très facile à visualiser. Cependant, dans l'analyse des résultats, il faut noter que la fonction objectif utilisée pour déterminer si la solution est bel et bien celle souhaitée (voir équation (113) et le chapitre Matériel et Méthode 3.3.1) donne des résultats oscillant entre 10^{-6} et 10^{-4}. Bien que la fonction objectif utilisée soit différente du critère d'arrêt de l'algorithme Hybride (équation(110)), tiré de la revue de littérature), les deux critères doivent tendre vers zéro lorsque la solution est trouvée. Un écart de près de six ordres de grandeurs entre la tolérance de l'algorithme Hybride et la fonction objectif signifie donc que le critère d'arrêt de l'algorithme Hybride n'est pas idéal pour le modèle ASM1 – Décanteur ponctuel.

Ces observations militent donc en faveur de plus de flexibilité au niveau des critères d'arrêt. De plus, compte tenu des faibles différences entre les résultats de l'algorithme de Broyden et de l'algorithme Hybride sur le modèle Prédateur – Proie et des performances inacceptables de l'algorithme de Broyden sur les autres modèles tests (Acide – Base et

ASM1 – Décanteur ponctuel), il a été décidé de travailler à partir de l'algorithme Hybride uniquement.

4.2.2.4 Discussion sur les résultats des algorithmes actuels

Les résultats présentés montrent bien la dépendance entre la solution d'un modèle au régime permanent et les estimés initiaux donnés à l'algorithme. Ils montrent également que de trouver le régime permanent d'un modèle à l'aide d'algorithmes spécialisés est plus complexe que d'utiliser une simulation sur une longue période en conservant les entrées constantes. En effet, les modèles Acide – Base et ASM1 – Décanteur ponctuel verront les résultats d'une simulation converger à leur seule solution analytique peu importe les valeurs initiales données au modèle, ce qui n'est pas le cas avec les algorithmes de calcul du régime permanent. Le modèle Prédateur – Proie convergera vers l'équilibre entre les populations en simulation de longue durée si les deux populations initiales sont non-nulles. Il faut donc des conditions particulières pour voir apparaître les autres solutions (population initiale de proies nulle pour converger à 0, 0 et population de prédateur nulle et de proie non-nulle pour converger à un équilibre chez les proies seulement).

Le critère d'arrêt de l'algorithme Hybride représente également un problème. La forme particulière du calcul de l'erreur d'une solution implique que des variables d'état présentant des valeurs très différentes nécessitent une tolérance très forte. Puisque plusieurs modèles peuvent apparaître en pratique, différents critères d'arrêt devraient être mis à la disposition de l'utilisateur. Ainsi, si les variables d'état sont toutes proches de zéro, un critère absolu permet généralement de mieux gérer l'erreur si une variable prend la valeur de zéro. À l'inverse, si les variables présentent des ordres de grandeurs très différents, un critère relatif permet de garantir un minimum de précision quel que soit l'ordre de grandeur d'une variable donnée. Il est à noter que l'algorithme de Broyden utilisé ne permet même pas le réglage de la tolérance, qui est gérée à l'interne.

Les modèles Acide – Base et ASM1 présentent un Jacobien singulier. Bien que cette singularité soit facile à identifier (variation nulle du débit d'eau par rapport aux autres variables d'état et elle-même dans le cas du modèle ASM1 et contrôle de la charge pour le modèle Acide - Base), l'algorithme de Broyden refuse de continuer le calcul. Bien que la

décomposition QR puisse être réalisée sur une matrice singulière, contrairement à la décomposition LU qui entraînera une division par zéro, l'algorithme de Broyden détecte un problème sous-déterminé et refuse de s'y attaquer. Dans le cas du modèle ASM1, la singularité interdit toute variation à la solution du volume d'eau puisque toutes les dérivées partielles sont nulles, donc à l'équilibre. Dans le cas du modèle Acide – Base, les dérivées partielles de la variable du contrôle de la charge ne sont pas toutes nulles. La variable peut donc varier et, dû à la singularité, sa valeur finale est aléatoire puisque le problème est sous-déterminé. Cependant, cette valeur finale n'est d'aucun intérêt. Tant que le reste du modèle est déterminé, la solution recherchée peut être atteinte. Un Jacobien singulier peut donc être un signe d'un modèle incomplet, mais également d'un comportement souhaité. L'algorithme utilisé pour calculer le régime permanent doit donc absolument pouvoir proposer une solution bien que le problème soit sous-déterminé.

4.2.3 Modifications du modèle pour favoriser la convergence vers la solution désirée

Le calcul du régime permanent d'un modèle par un algorithme tel que celui de Broyden ou l'algorithme Hybride reste une tâche incertaine comme l'ont montré les résultats des trois modèles tests (Acide – Base, Prédateur – Proie, ASM1 – Décanteur ponctuel). En effet, plusieurs solutions existent qui répondent au critère mathématique d'une dérivée nulle dans le temps, mais qui ne passent pas le test de la réalité (concentrations négatives, par exemple).

Cependant, le modélisateur possède une connaissance du comportement attendu de son modèle. C'est cette connaissance que la présente section injecte dans un modèle pour en favoriser la convergence à une solution répondant au critère mathématique, mais également à la réalité. Dans ce but, une modification des équations d'état est testée à la section 4.2.3.1 et la possibilité d'ajouter des contraintes dans le modèle est envisagée à la section 4.2.3.2.

4.2.3.1 Multiplication des équations d'état par une courbe de type gaussienne

Bien que la valeur à l'équilibre d'une variable d'état soit inconnue, le modélisateur a généralement une bonne idée de la zone dans laquelle devrait se trouver cette valeur. Dans

le but d'utiliser cette connaissance, chaque équation différentielle est multipliée par une fonction gaussienne modifiée. L'idée est d'imiter les fonctions de pénalités utilisées dans les algorithmes d'optimisation en introduisant un terme favorisant la convergence dans la zone souhaitée. En effet, puisque l'algorithme Hybride cherche à annuler les fonctions d'état (équations différentielles par rapport au temps) en utilisant les dérivées des variables d'état par rapport à elles-mêmes (utilisation du Jacobien), il serait possible de multiplier l'équation d'état par un terme non-nul très grand lorsque la variable d'état est hors de la zone souhaitée et très petit lorsque la variable d'état se retrouve dans la zone d'intérêt. Les essais ont été faits avec une fonction inspirée de la fonction gaussienne (équation (114)):

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{114}$$

Pour favoriser une zone sans créer de solutions supplémentaires, la fonction ne doit pas franchir zéro. Autrement, la multiplication de cette fonction par la fonction d'état à factoriser créerait de nouvelles racines. Dans ce but, le terme normalisant la fonction gaussienne est éliminé et l'exponentielle varie uniquement entre 0 et 1.

Cette équation est ensuite inversée pour tendre vers zéro dans la zone souhaitée tout en variant entre 0.1 et 1.1 :

$$f(x) = 1.1 - e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{115}$$

La valeur de 1.1 est arbitraire mais doit être choisie aussi proche de 1 que possible. Une valeur trop proche de 1, cependant, créera des minimums locaux, réduisant les performances des algorithmes d'autant. La moyenne μ centre le minimum de la courbe tandis que l'écart type σ sert à contrôler la largeur de la cloche. Une valeur de σ trop faible concentrera l'effet de la gaussienne à une zone très restreinte alors qu'une valeur trop élevée reviendra à diluer l'effet de l'ajout. En prenant $\mu = 0$ et $\sigma = 1$, la courbe de la Figure 28 est générée :

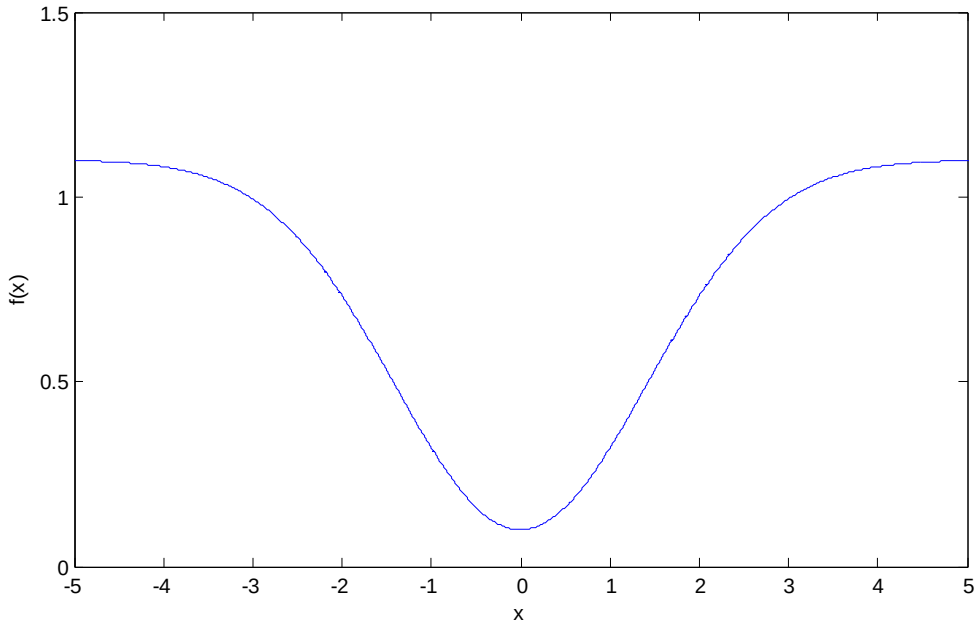


Figure 28 : Fonction Gaussienne inversée et modifiée

Lorsque la fonction dessinée à la Figure 28 est multipliée à une fonction d'état, la racine comprise dans la cloche sera favorisée. Pour le démontrer, considérons l'équation quadratique (116) dont les racines sont recherchées.

$$f(x) = 0{,}01 * (x + 3) * (x - 2) * (x - 10) \qquad (116)$$

Ces racines sont situées en -3, 2 et 10. Si la racine $x = 2$ est celle recherchée (par exemple, il s'agit de la seule valeur réaliste sur le système étudié), l'équation (116) sera multipliée par (115) avec les paramètres $\mu = 2$ et $\sigma = 3$. Il est possible d'estimer la zone de convergence des racines en fonction des extrema locaux de l'équation (116). En effet, un estimé initial convergera généralement vers la racine la plus proche à partir des extrema locaux. Les trois courbes sont représentées à la Figure 29.

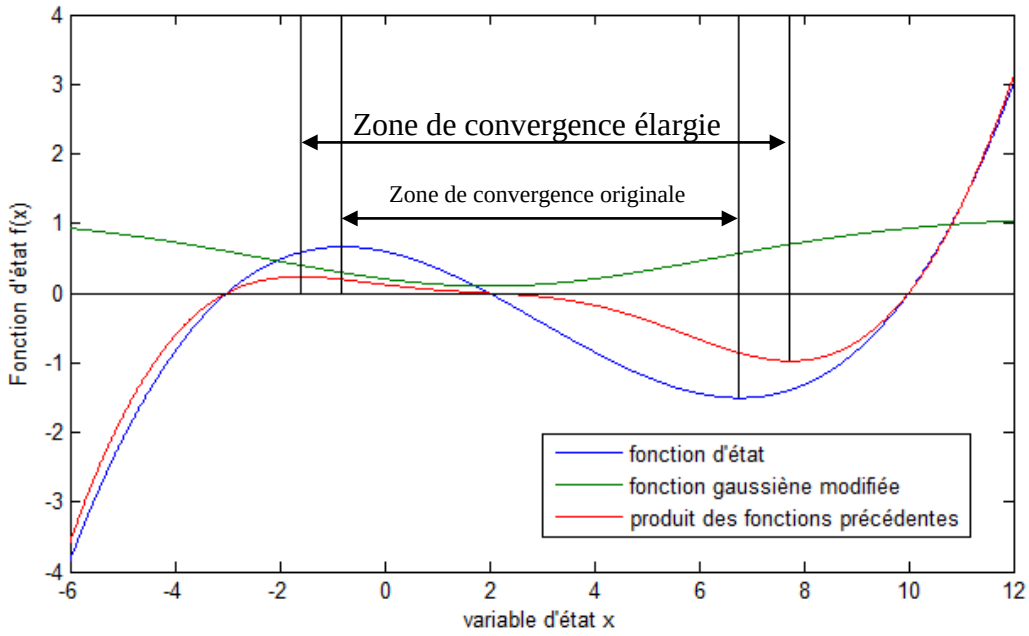


Figure 29 : Fonction d'état résultante avec une racine favorisée

Comme le montre la Figure 29, les trois racines existent toujours. Cependant, la plage dans laquelle l'estimé initial convergera vers la solution désirée augmente.

Cette technique a été testée sur le modèle Prédateur – Proie en utilisant les paramètres indiqués au Tableau 15 pour favoriser la convergence à des populations coexistantes (48 095 proies et 619 prédateurs). Ces paramètres ont été choisis de manière empirique pour offrir les meilleurs résultats possibles, résultats qui sont présentés à la Figure 30.

Tableau 15 : Paramètres de la courbe de type gaussienne pour le modèle Prédateur – Proie

Paramètre	Valeur
μ_{Proie}	50 000
σ_{Proie}	20 000
$\mu_{Prédateur}$	500
$\sigma_{Prédateur}$	200

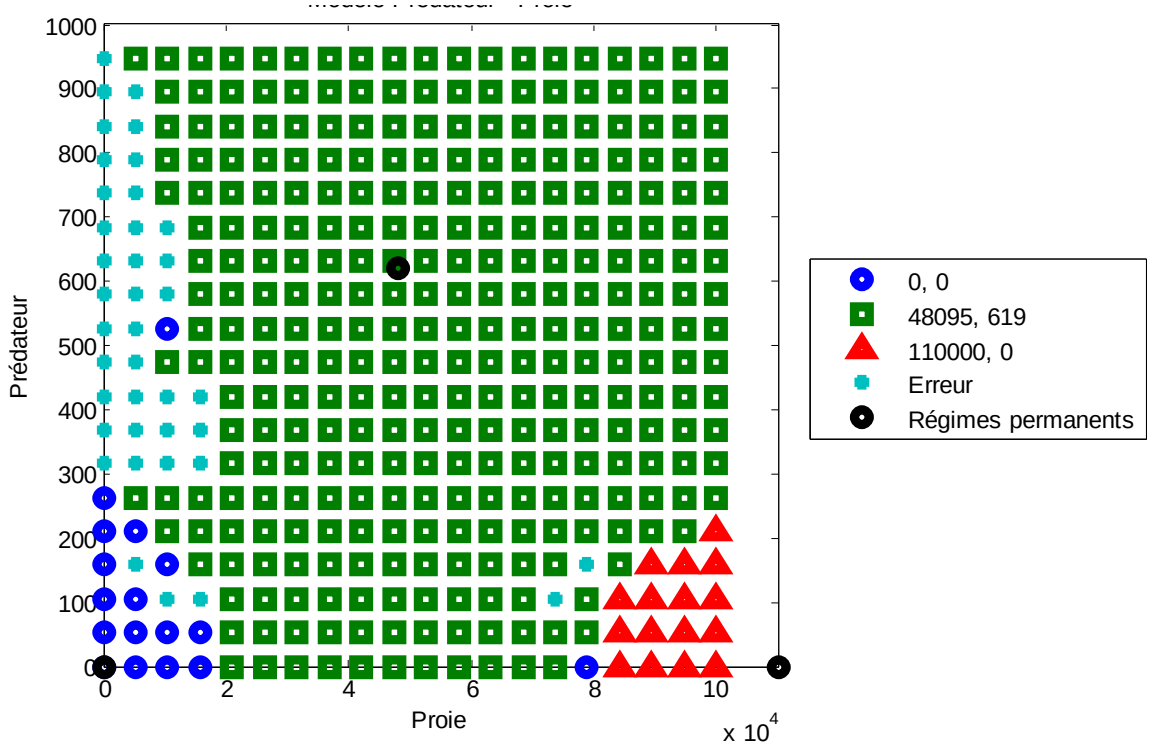


Figure 30 : Résultats de l'algorithme Hybride lorsque les équations d'état sont multipliées par une courbe en forme de cloche inversée

Les résultats de la Figure 30 se comparent avantageusement à ceux de la Figure 25. En effet, les zones d'attraction des racines indésirables sont réduites considérablement. Par contre, il est impossible de garantir que le régime permanent trouvé sera celui souhaité.

De plus, dans sa forme actuelle, la fonction de pénalité ne s'applique pas aux variables d'état pouvant varier sur plusieurs ordres de grandeurs comme les variables du modèle Acide – Base. L'équation (115) pourrait être modifiée en ce sens, mais la présence de racines indésirables et les nombreux paramètres à régler ne répondent pas à tous les besoins. C'est pourquoi, malgré des résultats supérieurs aux résultats originaux, cette approche est abandonnée et une technique éliminant complètement les résultats indésirables est recherchée.

4.2.3.2 Ajout de limites

Les valeurs limites que peuvent prendre les variables d'état peuvent déjà être incluses dans la formulation initiale d'un modèle dans le langage Modelica. Cette information sera, dans

cette section, utilisée pour garantir qu'une solution du calcul du régime permanent respecte ces limites.

4.2.3.2.1 Imposition d'un minimum ou d'un maximum

Il est possible de forcer l'algorithme de recherche du régime permanent à chercher dans une certaine plage de valeur en ajoutant une équation différentielle fictive dont la variable d'état n'est d'aucun intérêt, mais qui ne peut atteindre le régime permanent que si une variable d'état respecte ses limites. Ainsi, l'équation :

$$\frac{d(y_{fictive})}{dt} = y_i - (y_{i_{min}} + y_{fictive}^2) \tag{117}$$

où $y_{fictive}$ est une variable d'état fictive ajoutée, force y_i à converger vers une solution supérieure à $y_{i_{min}}$.

En effet, la solution à l'équation (117) est :

$$0 = y_i - (y_{i_{min}} + y_{fictive}^2) \tag{118}$$

$$y_{fictive} = \pm\sqrt{y_i - y_{i_{min}}} \tag{119}$$

Il existe donc deux solutions à l'équation (119) et l'une ou l'autre pourra être choisie sans influencer l'algorithme. Cependant, puisque les solutions complexes ne peuvent être proposées par l'algorithme, la variable d'état y_i doit obligatoirement être supérieure à $y_{i_{min}}$.

Cette dépendance se répercute dans le calcul du Jacobien puisque les termes $\frac{d(y_{fictive})}{d(y_i)}$ et $\frac{d(y_i)}{d(y_{fictive})}$ sont non-nuls. L'influence de l'équation (117) peut se visualiser aisément. Ainsi, si une équation quadratique (120) est prise comme test et qu'une racine positive ($\sqrt{5}$) est recherchée, le système d'équation (120) (121) peut être construit :

$$f_1(y_1, y_{fictive}) = y_1^2 - 5 \tag{120}$$

$$f_{limite}(y_1, y_{fictive}) = y_1 - (y_{1_{min}} + y_{fictive}^2) \tag{121}$$

Ce système d'équation peut être représenté en deux dimensions comme à la Figure 31.

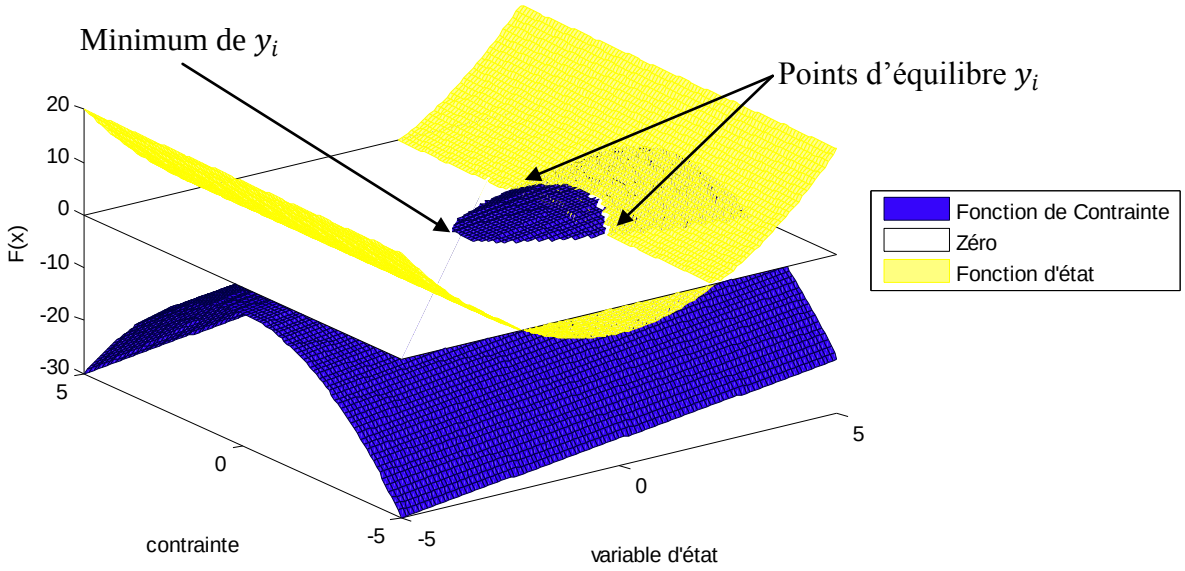


Figure 31 : Vue tridimensionnelle de l'influence d'une équation forçant la variable d'état à être au-delà d'un minimum

La Figure 31 montre que la variable d'état ne varie pas en fonction de la variable de contrainte ($y_{fictive}$). Par contre, la variable de contrainte ne croise la valeur zéro que lorsque la variable d'état est supérieure à son minimum (qui est 0). Sur la Figure 31, ce point correspond au passage au-dessus de zéro de la fonction de contrainte pour une valeur de la variable d'état. Une solution aux deux variables n'est donc possible que si la variable d'état respecte la contrainte. Dans le cas de la Figure 31, cette solution est l'un des deux points de jonction de la fonction d'état, la fonction de contrainte et de zéro. Il faut noter que la variable d'état recherchée est indépendante de la solution choisie.

S'il est possible de forcer une variable d'état à être supérieure à un minimum, l'inverse est également vrai. L'équation de contrainte devient simplement :

$$\frac{d(x_{N+1})}{dt} = x_i - (x_{i_{max}} - x_{N+1}^2) \qquad (122)$$

Ces équations de contrainte ont été ajoutées au modèle Prédateur – Proie pour forcer la convergence aux différentes solutions. Les limites utilisées sont arbitraires et choisies volontairement pour laisser beaucoup d'espace de recherche à l'algorithme :

$$\begin{aligned} Proie_{min} &= 1000 \\ Proie_{max} &= 80000 \\ Prédateur_{min} &= 100 \\ Prédateur_{max} &= 800 \end{aligned} \tag{123}$$

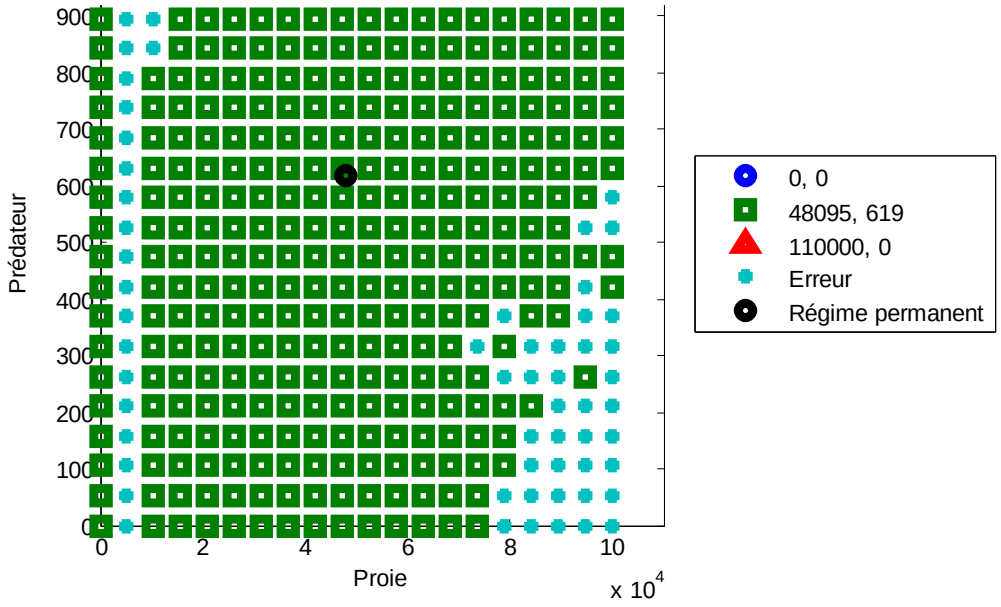


Figure 32 : Solutions en Régime Permanent pour le modèle Prédateur – Proie lorsque des limites sont imposées aux variables d'état.

Malheureusement, les estimés initiaux ne convergent pas tous à la solution désirée (Figure 32). Par contre, il n'y a aucune solution indésirable. Les régimes permanents en (0, 0) et en (110 000, 0) sont complètement éliminés, ce qui est très intéressant. Bref, cette technique permet d'interdire avec succès l'apparition de solutions indésirables. Le raisonnement peut être poussé plus loin si seules les solutions extrêmes sont désirées. La colonne d'erreur avec un estimé initial des Proies d'environ 5000 individus est intrigante. Bien que la raison exacte de ce résultat n'aie pas été recherchée, il est plus que probable qu'une difficulté numérique avec un tel estimé initial mène à une division par zéro ou un problème du même

genre. Ce cas montre bien l'intérêt de faire varier, même légèrement, les estimés initiaux lorsque le régime permanent est recherché à l'aide d'un algorithme spécialisé et qu'un message d'erreur est retourné.

Ainsi, pour converger à la solution (0, 0), une limite sur les populations à :

$$\begin{aligned} Proie_{max} &= 10000 \\ Prédateur_{max} &= 100 \end{aligned} \tag{124}$$

retourne les résultats de la Figure 33.

Encore une fois, plusieurs couples d'estimés initiaux ne convergent pas vers la solution. Par contre, l'algorithme détecte invariablement une mauvaise convergence plutôt que de retourner une solution violant la limite.

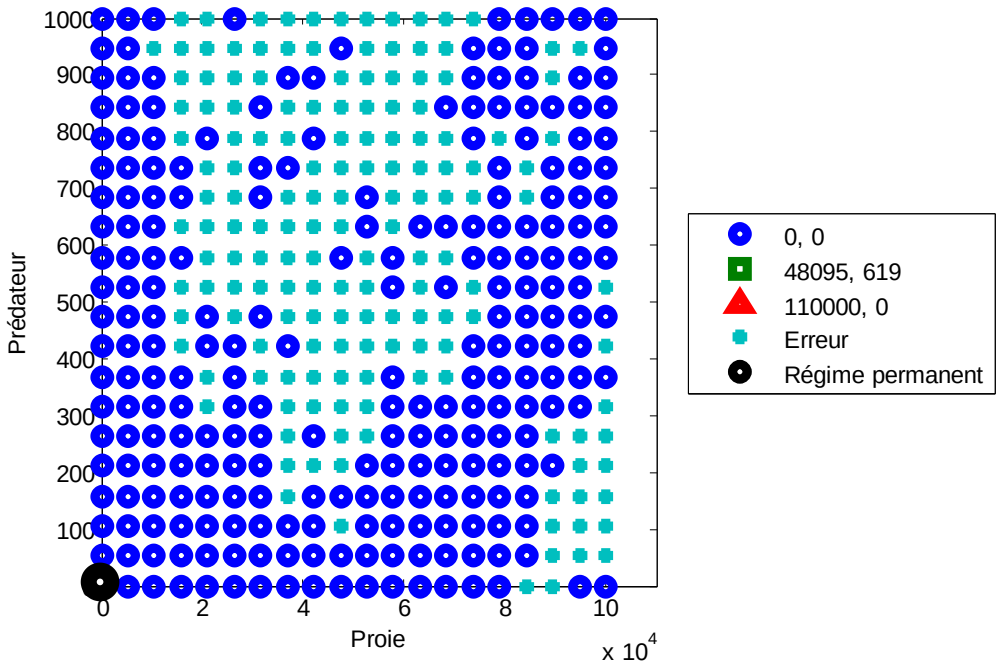


Figure 33 : Résultats du modèle Prédateur - Proie forçant la convergence à la racine (0, 0)

4.2.3.2.2 *Imposition d'un minimum et d'un maximum à l'aide d'une seule équation de contrainte*

Plusieurs variables d'état sont bornées par un minimum et un maximum, que ce soit pour offrir une signification physique ou simplement de l'information provenant de l'utilisateur

sur la plage de valeur normale de la variable. Cependant, comme la complexité du problème varie exponentiellement avec le nombre d'équations à résoudre simultanément, il peut être embêtant de multiplier le nombre d'équations différentielles à résoudre par trois (l'équation, sa contrainte minimale et sa contrainte maximale exprimées par les équations de la section précédente). Dans le but de réduire le nombre d'équations à résoudre, il est possible de construire une seule équation contraignant sa variable d'état associée entre un minimum et un maximum. Pour ce faire, l'équation de la contrainte est composée à la base d'une fonction arctangente. Ainsi, une équation de limite de forme équivalente à (117) est présentée à l'équation (125) :

$$der(x_{fictive}) = x_i - \left(\frac{(C_{max} - C_{min})}{2} * \left(\frac{\arctan(x_{fictive})}{\pi/2} + 1 \right) + C_{min} \right) \qquad (125)$$

La forme (125) permet d'éliminer toute racine située à l'extérieure des limites comme le montre la Figure 34. Le modèle Prédateur – Proie est utilisé pour démontrer l'effet des limites avec une fonction arctangente. Les limites imposées sont les mêmes qu'à l'exemple précédent (123).

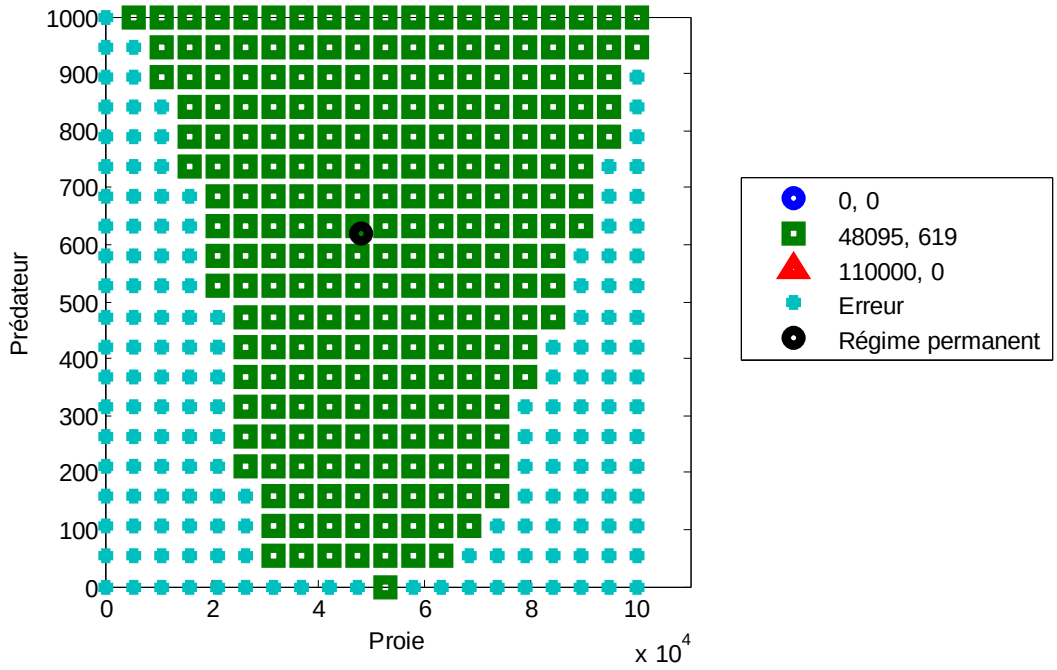


Figure 34 : Régime permanent du modèle Prédateur – Proie avec limites de type arctangente.

Bien que satisfaisants, les résultats illustrés à la Figure 34 sont loin d'être idéaux. En effet, une modification simple de l'équation (125) permet d'améliorer sensiblement les résultats. Il suffit en effet de normaliser $x_{fictive}$ par l'intervalle à couvrir comme suit :

$$der(x_{fictive}) = x_i - \left(\frac{(C_{max} - C_{min})}{2} * \left(\frac{\arctan\left(\frac{x_{fictive}}{(C_{max} - C_{min})} \right)}{\pi/2} + 1 \right) + C_{min} \right) \qquad (126)$$

Cette modification permet de recalculer les résultats présentés à la Figure 34 et donne les résultats de la Figure 35.

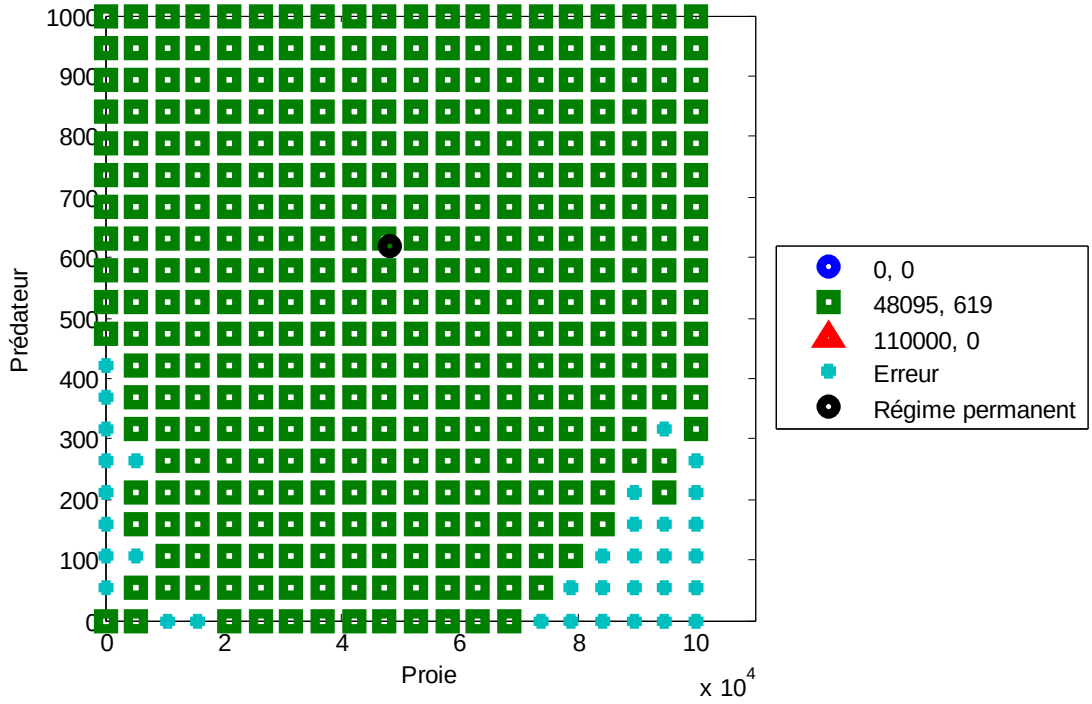


Figure 35 : Régime permanent du modèle Prédateur – Proie avec limites de type arctangente et variable fictive normalisée par l'intervalle à couvrir

Bien qu'analytiquement, les équations (125) et (126) soient aussi difficiles à résoudre, des questions de stabilité numérique privilégient l'utilisation de (126) pour résoudre un modèle au régime permanent avec limites. En effet, les plages de variation des variables d'état sont similaires dans le cas de (126). Ainsi, des corrections de magnitude similaire sont obtenues à chaque pas de l'algorithme, ce qui réduit les erreurs numériques et les erreurs

d'interprétation comme une fausse convergence d'une variable variant beaucoup moins qu'une autre. Ainsi, la variable fictive associée à la population de proies possède une valeur finale de 0.18 lorsque l'équation (125) est utilisée et de 13004 lorsque l'équation (126) est utilisée. Puisque la population de proies se stabilise à 48095 individus, la variable fictive diffère de moins d'un ordre de grandeur contre plus de 5 ordres de grandeur lorsque la limite est définie par l'équation (125).

Quant à comparer les résultats obtenus à la Figure 32 par rapport à ceux de la Figure 35, les taux de convergence sont très similaires, bien que la Figure 32 semble moins stable numériquement puisqu'un estimé initial des proies à 5 000 individus ne peut converger peu importe l'estimé initial de la population des prédateurs. Cela dit, le principal avantage d'utiliser l'expression (126) pour contraindre la recherche réside dans le nombre d'équations algébriques implicites à ajouter au modèle.

4.2.3.2.3 Résultats sur le modèle ASM1

L'ajout de limites sur le modèle ASM1 se fait sur la base d'information connue. En effet, puisque le modèle ASM1 a été publié en 1987 (Henze et al. 1987), ses utilisateurs connaissent généralement une fourchette dans laquelle doivent se situer les variables d'état. Cette connaissance du modèle permet de fournir les limites inférieures et supérieures aux variables d'état (Tableau 16).

Les résultats du modèle ASM1 – Décanteur ponctuel sont calculés avec les limites de la même façon que pour le modèle original (voir Figure 27). Les résultats sont présentés à la Figure 36.

Tableau 16 : Limites inférieures et supérieures des variables d'état du modèle ASM1 – Décanteur ponctuel

Variable	Limite inférieure (g)	Limite supérieure (g)	Limite inférieure (g/m^3)	Limite supérieure (g/m^3)
S_S	10	100 000	0,01	100
S_I	1000	50 000	1	50
S_NH	10	50 000	0,01	50
S_ND	10	10 000	0,01	10
S_NO	10	50 000	0,01	50
S_O	10	10 000	0,01	10
S_ALK	1000	100 000	1	100
X_BA	1000	500 000	1	500
X_BH	10 000	5 000 000	10	5 000
X_I	10 000	5 000 000	10	5 000
X_ND	1	500 000	0,001	500
X_P	100	5 000 000	0,1	5 000
X_S	100	1 000 000	0,1	1 000

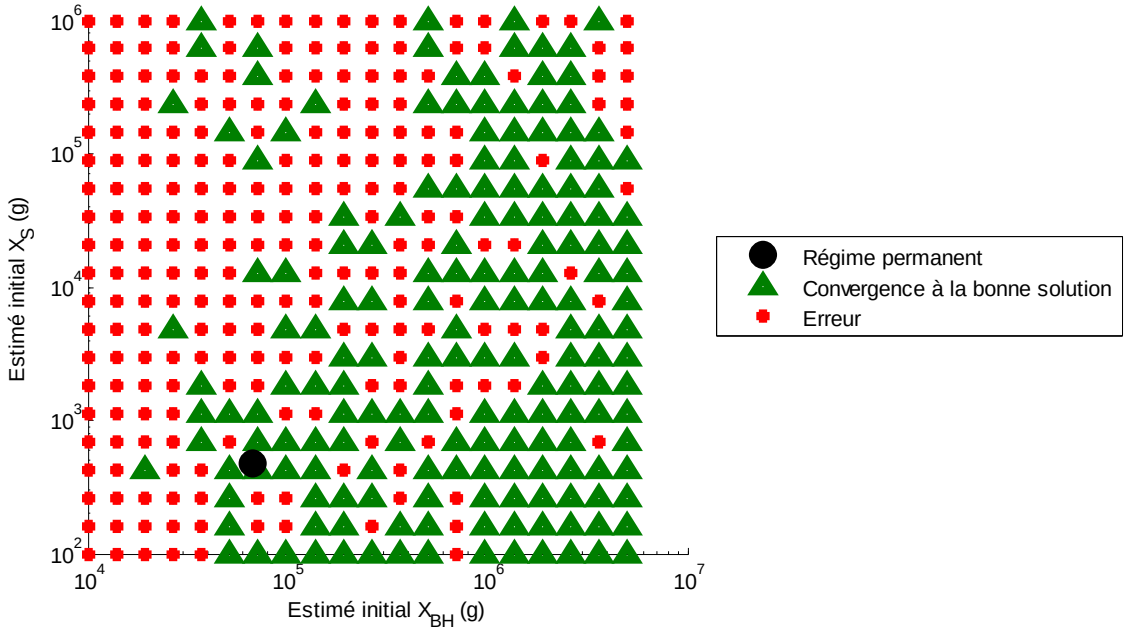


Figure 36 : Résultats du calcul du régime permanent sur le modèle ASM1 – Décanteur ponctuel avec limites

La Figure 36 montre que l'algorithme converge moins souvent à la bonne solution que lorsqu'aucune limite n'est imposée et que la zone de convergence est moins clairement

définie. Cependant, la Figure 27 montre bien que la majorité des estimés initiaux convergent vers une solution indésirable (possédant des concentrations négatives) alors qu'une solution calculée avec des limites est incapable de converger à l'extérieur de celles-ci. L'utilisation de limites génère donc un modèle plus complexe à résoudre, mais dont les seules solutions possible sont à l'intérieur des bornes imposées.

4.2.3.2.4 Résultats sur le modèle Acide – Base

Le modèle Acide – Base a comme caractéristique de laisser varier ses variables d'état sur une très grande amplitude de par la nature logarithmique des concentrations des ions. Dans ces conditions, la construction de limites sous forme de fonctions arctangente (avec des concentrations minimale de zéro et maximale de 1) empêche l'algorithme de converger. En effet, la mise à l'échelle se révèle inadéquate pour une telle fonction puisque la solution désirable de S_H est $1{,}27 \times 10^{-7}$ et que la fonction de limite associée cherche une solution entre les valeurs 1×10^{-14} et 1, soit 14 ordres de grandeur de différence.

Par contre, il est toujours possible de n'imposer qu'une limite inférieure à zéro (équation (117)). Dans ces conditions, selon les équations proposées, les variables fictives associées aux trois variables d'état ont une valeur finale de l'ordre de 10^{-4}, soit des valeurs assez proches des solutions recherchées pour atteindre la solution globale du modèle. Les résultats de la recherche du régime permanent pour des variables d'état strictement positives sont affichés à la Figure 37.

Ces résultats montrent que l'ajout de limite permet une convergence presque systématique à la solution recherchée et sont donc définitivement plus intéressants que ceux trouvés à l'aide du modèle sans limite (voir Figure 26).

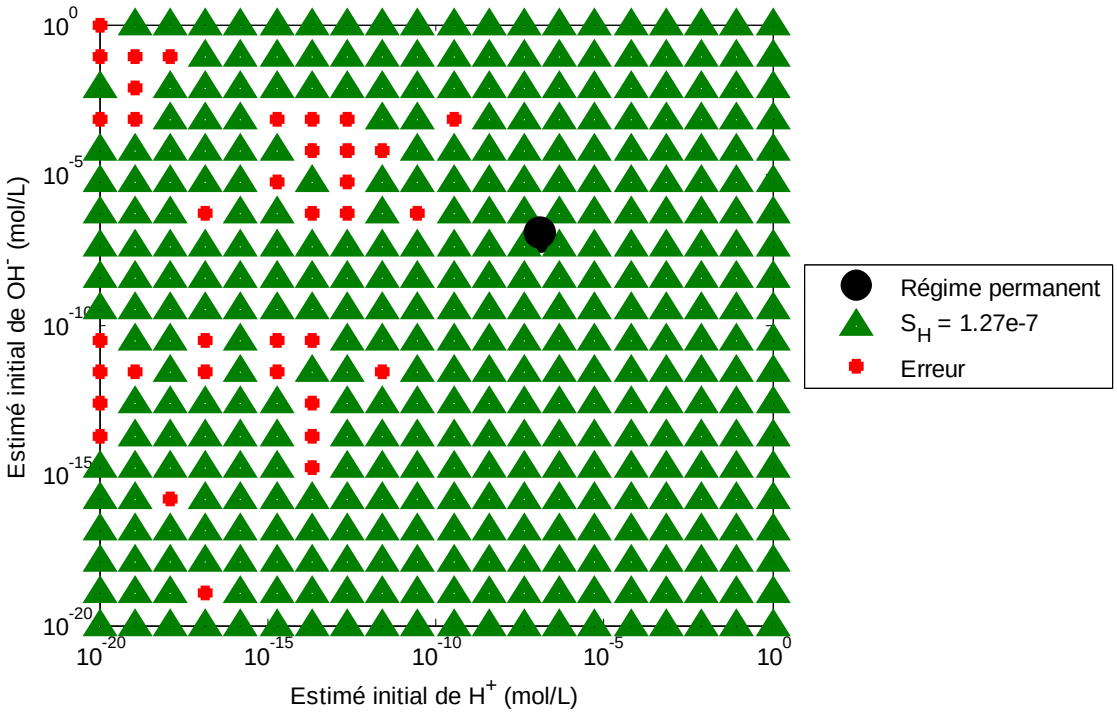


Figure 37 : Solution au régime permanent du modèle Acide – Base lorsque les variables d'état sont contraintes à être positives.

4.2.3.2.5 Conclusions sur l'utilité des limites

Les résultats obtenus à l'aide de limites sous la forme d'équations algébriques à résoudre simultanément avec les équations différentielles sont probant : Aucune solution n'est possible hors des limites.

Il est important, par contre, de réaliser que l'utilisation de limites sous forme d'équation algébrique implicite doit être utilisée avec jugement. En effet, si chaque variable différentielle possède des limites inférieures et / ou supérieures, la création d'équations supplémentaires vient doubler le nombre d'équations à résoudre. Les formes d'équations proposées se résolvent très bien par les algorithmes disponibles : elles ne présentent aucune discontinuité, sont très lisses et ne génèrent aucun problème particulier. Cependant, le calcul du Jacobien exige le double d'évaluations du modèle pour une matrice finale quatre fois plus grande. De plus, la solution de sa factorisation varie selon le nombre de variables au cube.

Les connaissances à priori du modèle permettent d'inclure plus que simplement des bornes sur les valeurs normales du modèle. Le modèle ASM1, par exemple, ne consomme ni ne produit d'éléments inertes (solubles ou particulaires). Autrement dit, au régime permanent, la concentration en éléments inertes à la sortie du réacteur sera identique à celle à l'entrée. Il n'est donc pas pertinent d'ajouter une équation de limite pour ces variables d'état. Dans le modèle ASM1 – Décanteur ponctuel, trois variables d'état répondent à cette définition, soit le volume d'eau dans le réacteur et les concentrations en composés inertes solubles et particulaires.

4.2.3.3 Génération automatique de limite sous Tornado

Les sections précédentes ont démontrées que l'imposition de limites sous forme d'équation algébrique implicite permet d'interdire la convergence d'une équation différentielle à l'extérieur de ces limites. Cependant, la génération de ces limites ne doit pas être une tâche qui revient au modélisateur. Le rôle du modélisateur doit rester d'imposer des minimums et maximums cohérents. C'est pourquoi, dans l'environnement Tornado, une fonction a été ajoutée pour générer ces équations automatiquement et pour les ajouter au modèle quand le régime permanent est recherché.

Cette fonction est appelée lors de la compilation du modèle par le programme MOF2T. À ce moment, la fonction de génération automatique de limites parcourt la liste de variables pour repérer les variables d'état. Cette opération est réalisée dans une liste de tous les symboles du modèle (nom des variables, des paramètres, des fonctions, etc.) et leur description. Une fois que toutes les variables d'état sont identifiées, les minimums et maximums de chacune d'entre elles sont contrôlés. En effet, sous Tornado, toutes les variables possèdent un minimum et un maximum. Lorsque le modélisateur n'en spécifie aucun, la valeur par défaut est –INF ou +INF, selon le cas. Il est donc possible de détecter les variables pour lesquelles des limites sont imposées en cherchant des limites différentes de –INF et +INF.

Si les deux limites existent pour une variable, une équation de contrainte sous la forme (126) est ajoutée aux équations différentielles. Si une seule limite existe (minimum ou maximum), la limite correspondante (équation (117) ou (122)) est générée.

La fonction de génération automatique des contraintes est donnée à l'Annexe C.

4.2.3.4 Discussions sur l'ajout de limites

Parmi les éléments améliorables de la génération automatique de limites, il y a l'usage de quantités numériques. En effet, dans sa version actuelle, Tornado assigne un maximum et un minimum numérique à toutes les variables. Il est donc impossible de construire une équation de limites sur la base d'une variable y_{min} ou y_{max}. Seule une valeur numérique peut être utilisée. L'inconvénient principal de ce comportement est l'impossibilité de modifier les équations des limites après la compilation du modèle.

Un cas typique où ce comportement pourrait être problématique est une variable comportant des racines multiples. Dans ce cas, une approche logique serait de réduire la zone permise à cette variable d'état pour la forcer à converger vers une solution particulière. Malheureusement, dans sa forme actuelle, Tornado ne permet pas une telle stratégie. Les limites sont en effet fixées au moment de l'écriture du modèle. Dans le cadre du logiciel de simulation WEST, qui, à terme, servira d'interface entre l'utilisateur et Tornado, ces limites ne peuvent plus être changées au moment de la configuration du modèle complet. Les équations de limites sont donc créées au moment de la compilation du modèle avec des valeurs numériques plutôt que des variables modifiables à posteriori.

Néanmoins, l'utilisation de limites présente tout de même des avantages à l'utilisation du modèle original. L'avantage le plus important est l'absence de convergence hors de la zone permise. Considérant qu'une réponse erronée est plus problématique qu'un échec de l'algorithme, l'ajout de limites à un modèle permet de garantir des solutions ayant un sens pour l'utilisateur ou un message d'erreur. Dans ces conditions, il est possible de construire des stratégies de recherche du régime permanent en perturbant les estimés initiaux a répétition jusqu'à ce qu'une solution soit trouvée. Compte tenu du faible coût de calcul de la recherche du régime permanent avec l'algorithme Hybride par rapport au temps

nécessaire pour réaliser une simulation dynamique jusqu'à l'équilibre, une telle solution pourrait offrir d'excellents résultats.

Dans sa version actuelle, les limites sont intégrées au modèle au moment de la compilation. Ce faisant, un modèle généré pour la recherche du régime permanent ne peut être utilisé en simulation dynamique puisque les équations de limites sont des équations algébriques implicites construites sous la forme d'équations différentielles. Lors du calcul du régime permanent, ces deux formes sont équivalentes, mais garder les équations de contrainte en simulation dynamique revient à ajouter des équations différentielles instables (dont la variable d'état varie de façon incontrôlable). Si un modèle doit servir en régime permanent et en régime dynamique, il faut soit compiler le modèle original pour chaque usage, soit renoncer aux limites imposées sur les variables d'état. Cependant, compte-tenu de la forme standardisée des équations de limites, il serait possible d'inclure leur gestion à l'intérieur même d'un algorithme de calcul du régime permanent sans avoir à les compiler à priori dans le modèle. Malheureusement, la forme actuelle de Tornado ne permet pas une telle gestion puisqu'il n'est pas possible d'avoir accès aux valeurs minimales et maximales de chaque variable.

4.3 Dérivation symbolique sur Tornado

L'outil de base des algorithmes de calcul du régime permanent présentés dans ce mémoire, soit l'algorithme de Broyden et l'algorithme Hybride est le Jacobien. Ce même Jacobien est également la base de l'algorithme d'intégration numérique DIRK présenté à la section 4.1.1 de même que de la technique de réduction automatique de modèle.

Si un calcul numérique est généralement satisfaisant, ce calcul présente des risques d'imprécision et peut dans certains cas être très gourmand en temps de calcul (De Pauw et Vanrolleghem 2006). Par exemple, le calcul du Jacobien de l'algorithme de Broyden est réalisé par différences finies où la perturbation est de 0,01%. Lorsque la variable d'état est nulle, la perturbation est de 10^{-4}. Une perturbation relative de 0,01% sera dans la très grande majorité des cas suffisamment précise, bien que des équations très non-linéaires puissent être très mal gérées (exemple : une fonction sinusoïdale à très haute fréquence). Une perturbation absolue, pour sa part, est encore plus risquée. En effet, si les variables d'état affectées par la variable perturbée sont trop grandes, la perturbation risque de disparaître dans l'erreur sur la variable et ne pas retourner le résultat désiré. Mais l'erreur est au moins aussi importante, sinon plus, lorsque la variable possède une plage de variation restreinte comme, par exemple pour un calcul de pH. Dans ce cas, si l'équilibre est autours de pH 7, une perturbation absolue de 10^{-4} correspond à 1000 fois la valeur d'équilibre.

Ces deux exemples viennent donc démontrer l'intérêt de réaliser une différentiation symbolique pour un logiciel de simulation qui souhaite être le plus général possible. Finalement, des raisons de performance peuvent être invoquées pour justifier la dérivation symbolique. En effet, une dérivation numérique requière $N + 1$ appels au modèle. Or, dans le cas de plusieurs modèles de grande envergure, un nombre très important d'éléments du Jacobien sont nuls et le resteront (par exemple, la variation du volume d'eau en fonction de la variation de l'oxygène dissous). Une dérivation symbolique permettrait de poser toutes ces dérivées partielles à zéro et de ne jamais plus tenter de les calculer. La Figure 38 montre le Jacobien du modèle BSM1 où les éléments non-nuls de la matrice sont représentés par un

point et les éléments nuls ne sont pas affichés. Il est possible d'y identifier les différents éléments du modèle BSM1.

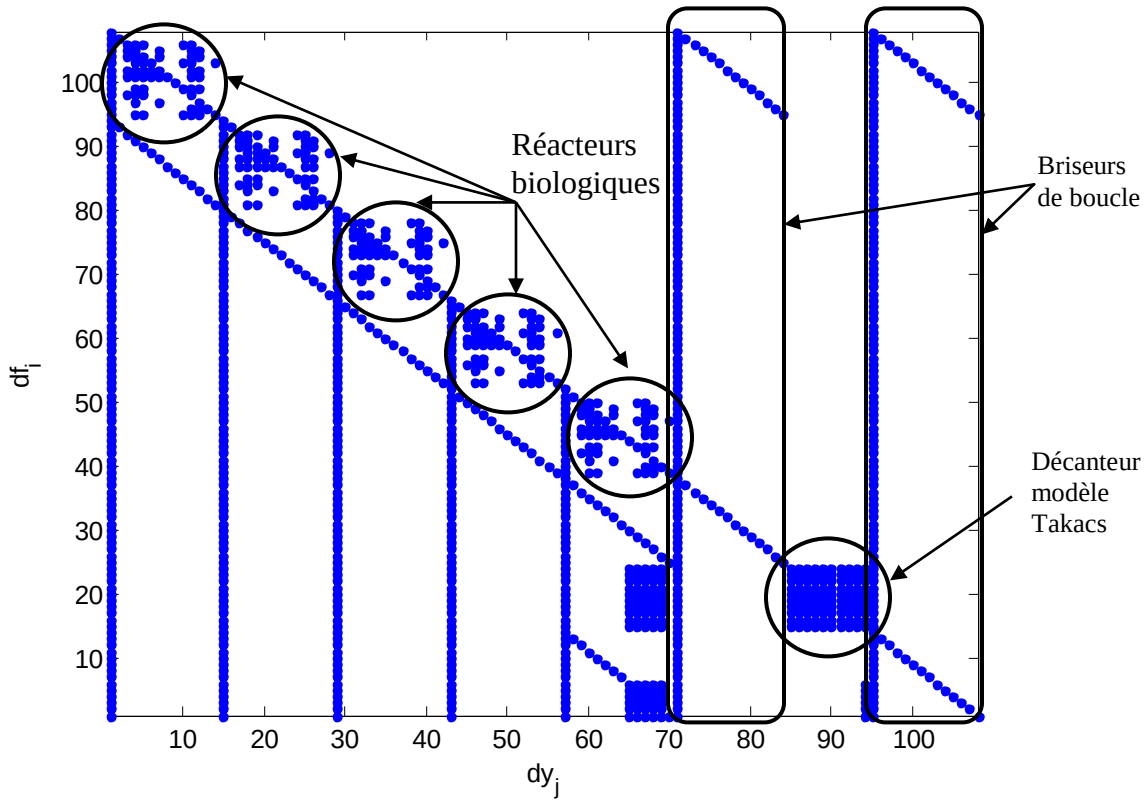


Figure 38 : Représentation graphique du Jacobien symbolique du modèle Benchmark

Ce que la Figure 38 démontre c'est que pour un modèle de 108 variables d'état, à peine 10% des éléments du Jacobien sont non-nuls (1227 / 11664). Une telle figure est également représentative de la modélisation de haut-niveau puisque de tels modèles sont construits à partir de blocs plus petits. Ainsi, ces sous-modèles présentent de très fortes interdépendances mais sont quasiment indépendants des autres sous-modèles (à quelques exceptions près bien visibles sur la figure).

Puisqu'un Jacobien numérique se calcule colonne par colonne, la Figure 38 montre que la réalisation d'un Jacobien hybride, c'est-à-dire dont les éléments isolés sont calculés symboliquement et les colonnes très denses sont calculées numériquement, pourrait offrir un compromis très intéressant entre performances en évitant d'avoir à recalculer le modèle en entier pour deux ou trois éléments de la matrice et en évitant des calculs complexes

lorsqu'une seule évaluation du modèle supplémentaire permet d'obtenir des approximations intéressantes. Ainsi, si seules les colonnes contenant moins de 50 éléments non-nuls étaient calculées symboliquement, 611 éléments doivent être dérivés. Les 616 restants peuvent être calculés à l'aide de 7 différentiations numériques (contre 108 si le Jacobien entier est calculé numériquement).

4.3.1 Principes de base de la dérivation symbolique

La dérivation symbolique est réalisée au moment de la conversion du modèle en langage de modélisation (Modelica, MSL ou autre) vers le langage C. À ce moment, le modèle est interprété par le compilateur MOF2T et est représenté par un arbre tel qu'illustré dans le chapitre Matériel et Méthodes. Cette forme de représentation est parfaite pour la dérivation parce que cette dernière est très simple à implémenter sur une base récursive. L'architecture de la fonction de dérivation comprend une fonction maîtresse qui lit le prochain terme à dériver. Selon la nature de ce terme (addition, soustraction, nombre, variable, etc.), une sous-fonction appropriée est appelée.

Les termes qui doivent être dérivés sont décrits au Tableau 17.

Tableau 17 : Nœuds qui doivent être dérivables

IF	Condition If – Else – Then
ELSEIF	Conditions imbriquées
OR	Opérateur logique OU
AND	Opérateur logique ET
NOT	Opérateur logique NON
LT	Test Est Plus petit que
LE	Test Est Plus petit ou égal à
GT	Test Est Plus grand que
GE	Test Est Plus Grand ou égal à
EQ	Test Est Exactement
NEQ	Test Est Différent de
UPLUS	Plus unaire
UMINUS	Moins unaire
PLUS	Addition de deux expressions
MINUS	Soustraction de deux expressions
MUL	Multiplication de deux expressions
DIV	Division entre deux expressions
POW	Puissance de deux expressions
FUNC	Fonction d'une ou deux expressions
NUMBER	Nombre
BOOLEAN	Variable booléenne Vrai – Faux
ID	Variable

Chaque élément du Tableau 17 possède une fonction correspondante pour dériver l'expression en question. La fonction suivante est un exemple de la dérivation d'une addition :

```
CASTNode* CASTDerive::
DerivePLUS(CASTNode* pTerm)
{
  //Déclaration et assignation de variables
  CASTNode* pChild1;
  CASTNode* pChild2;

  pChild1 = pTerm->GetChildren().at(0);
  pChild2 = pTerm->GetChildren().at(1);

  //Dérivation des expressions
  auto_ptr<CASTNode> pDer1(Derive(pChild1));
  auto_ptr<CASTNode> pDer2(Derive(pChild2));

  //Gestion des dérivées
  if((pDer1->GetType() == CASTNode::EMPTY) &&
     (pDer2->GetType() == CASTNode::EMPTY))
    return m_pRoot->CreateEMPTY();

  else if(pDer1->GetType() == CASTNode::EMPTY)
    return pDer2.release();

  else if(pDer2->GetType() == CASTNode::EMPTY)
    return pDer1.release();

  else
    return m_pRoot->CreatePLUS(pDer1.release(),
                               pDer2.release());
}
```

Dans le code présenté, les objets CASTNode sont des nœuds de la classe AST (« Abstract Syntax Tree », ou Arbre Syntaxique Abstrait). Il s'agit de l'unité de base d'information du modèle et contient les informations suivantes :

- Type de nœud
- Lien vers d'autres nœuds (enfants)
- Information sous forme de nombre, suite de caractère ou variable booléenne (vrai ou faux)

Cette fonction est appelée lorsqu'une addition doit être dérivée. Une fois ses objets internes déclarés et assignés, elle dérive les expressions à additionner. Puisque la fonction ignore la nature de ces termes, elle les dérive à l'aide de la fonction générique chargée de déterminer avec quelle sous-fonction dériver ces termes (Fonction `Derive(pChild_)`). Finalement, les dérivées des expressions doivent être analysées pour gérer les retours nuls définis par le

nœud vide (EMPTY). La solution algébrique est retournée sous forme de nœud qui possède l'information de la dérivée algébrique.

La gestion des nœuds nuls est essentielle puisque la réécriture du modèle de sa forme abstraite (sous forme de nœuds) à un langage comme le C ne tient pas compte des nœuds nuls. Cette gestion permet également de réduire la taille de la représentation abstraite du modèle lorsque, par exemple, une multiplication par zéro apparaît.

Toutes les fonctions de dérivations en appellent une ou plusieurs autres à deux exceptions près. La première est la dérivée d'un nombre qui retourne simplement un nœud vide (la dérivée d'une constante est zéro). La seconde est la dérivée d'une variable. Celle-ci sera comparée à la variable par rapport à laquelle l'équation est dérivée. S'il s'agit de la bonne variable, le nombre 1 est retourné. Dans le cas contraire, l'équation de la variable est recherchée dans la liste des équations. Si cette équation est trouvée, elle est dérivée à son tour. Sinon, cette variable est considérée comme étant un paramètre et dérivé comme tel (un paramètre étant constant par définition, sa dérivée est nulle).

4.3.2 Dérivation de fonctions non-analytiques

Les langages de programmation, que ce soit Modelica, MSL, C++ ou tout autre langage, supportent des fonctions non-analytiques dont les dérivées ne se retrouvent dans aucune table. Ces nouvelles fonctions doivent donc être dérivées selon de nouvelles bases.

4.3.2.1 Instruction Conditionnelle

La première classe de fonctions non-analytique correspond aux instructions conditionnelles mieux connues sous le titre de If – Then – Else. Cette fonction sera analysée à l'aide de l'exemple suivant :

$$\begin{aligned} Si\ (x < 2{,}5)\ alors\ f(x) &= x^2 \\ Sinon\ f(x) &= x \end{aligned} \qquad (127)$$

L'équation (127) est un exemple simple, mais qui peut être généralisé à tous les cas possibles en Modelica. En effet, dans ce langage de programmation, l'instruction

conditionnelle correspond à l'opérateur ternaire du C++ « ? » et doit être utilisé comme suit :

$$X = (\text{condition}) \ ? \ (\text{alors } \ldots) : (\text{sinon } \ldots); \qquad (128)$$

Autrement dit, si la condition est vraie, l'expression (alors …) est exécutée et la réponse est transmise à X. Dans le cas contraire, c'est l'expression (sinon …) qui l'est.

Lorsqu'une expression de la forme de (127) doit être dérivée, seules les expressions (alors ..) et (sinon …) doivent être dérivées. La condition est reproduite exactement. Ainsi, l'équation (127) peut être tracée avec sa dérivée comme suit :

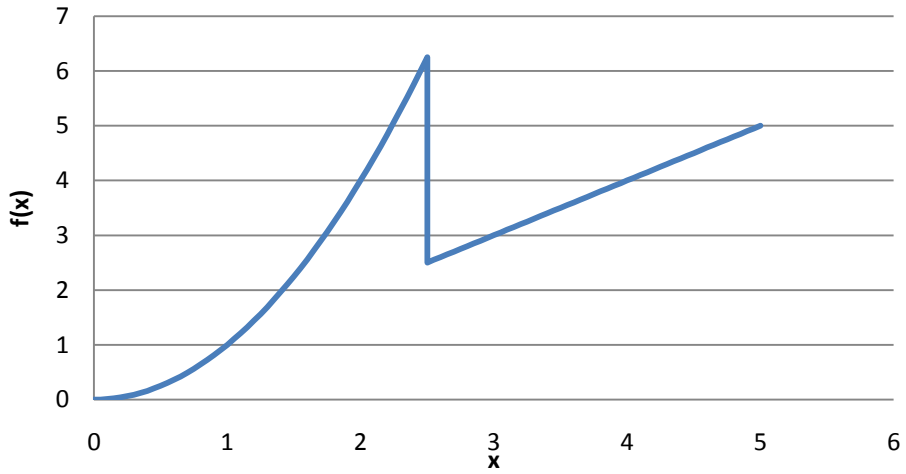


Figure 39 : Courbe de l'équation (127)

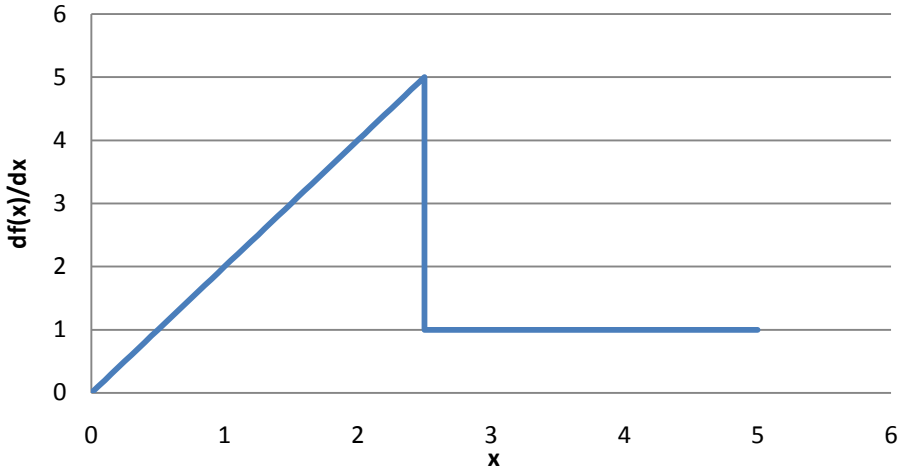


Figure 40 : Courbe de la dérivée de l'équation (127)

Ces deux figures montrent bien que la condition ne doit pas être modifiée si l'équation est dérivée. Il reste cependant une discontinuité sur le point de la condition. Cette discontinuité ne peut s'exprimer mathématiquement avec la méthode proposée. Cependant, comme la condition n'est pas modifiée, la solution sur la discontinuité sera donnée directement. Ainsi, dans l'exemple (127), si $x = 2{,}5$, la condition est fausse et sa dérivée vaut 1. D'un point de vue de continuité, cette approche se compare avantageusement à la dérivation par différences finies puisque son comportement ne génère pas de pente qui tend vers l'infini. De plus, en pratique, il est très rare de voir une variable se maintenir sur la limite de la condition puisque les variables entières ne peuvent pas être utilisées dans les langages de modélisations utilisés par la plateforme Tornado. Ce faisant, les variables réelles seront généralement à quelque distance de la discontinuité.

4.3.2.2 Fonction ATAN2

La fonction ATAN2 est identique à la fonction arc tangente au détail près qu'elle nécessite deux arguments x et y plutôt qu'un seul pour ATAN, ϕ. Le fait d'avoir deux arguments permet de déterminer le signe de la réponse en fonction du quadrant dans lequel se retrouvent x et y. La bonne nouvelle est que la dérivée de cette fonction est la même que celle de la fonction ATAN pour laquelle $\phi = \frac{y}{x}$.

4.3.2.3 Fonction PREVIOUS

La fonction previous(x) n'a un sens qu'en simulation dynamique où elle retourne la valeur de x calculée au dernier point d'intégration. Puisqu'en simulation dynamique ce dernier point d'intégration est à un temps très proche du point en cours, une erreur très faible est induite si sa dérivée est calculée au point actuel. Néanmoins, la dérivée formelle au temps présent est la dérivée de l'expression au temps d'intégration précédent.

$$y = previous(u) \tag{129}$$

$$\frac{dy}{dx} = \frac{d\big(previous(u)\big)}{dx} = previous\left(\frac{du}{dx}\right) \tag{130}$$

4.3.2.4 Fonction Valeur Absolue

La dérivée d'une valeur absolue est déterminée par une instruction conditionnelle. Si l'expression originale est positive, la dérivée originale est retournée. Dans le cas contraire, la dérivée est multipliée par -1. Le cas où $f(x) = 0$ est traité comme étant supérieur à zéro. Bien que le formalisme mathématique ne le permette pas puisqu'il s'agit d'une discontinuité et que zéro n'est ni positif ni négatif, une telle discontinuité ne peut être représentée numériquement. Cette décision se compare d'ailleurs avantageusement à la dérivée calculée par différences finies qui donne dans tous les cas une dérivée variant entre la dérivée donnée par une expression initialement positive et celle donnée par une valeur initialement négative. À défaut de traiter formellement le cas $f(x) = 0$, la discontinuité dans la dérivée apparaît, ce qui est souhaitable.

4.3.2.5 Fonction DELAY

La fonction DELAY correspond à retourner la valeur d'une expression u au temps $t - \delta$. Dans ces conditions, la dérivée de u est calculée comme suit :

$$y = delay(u, \delta) \tag{131}$$

$$\frac{d\big(delay(u, \delta)\big)}{dx} = delay\left(\frac{du}{dx}, \delta\right) \tag{132}$$

4.3.2.6 Fonction Max et Min

Les dérivées des fonctions Maximum et Minimum doivent tenir compte des expressions originales. Elles sont donc construites sous forme d'une instruction conditionnelle comme suit :

$$y = \max(v, u) \tag{133}$$

$$\frac{dy}{dx} = Si\ (v \geq u)\ Alors\left(\frac{dv}{dx}\right), Sinon\left(\frac{du}{dx}\right) \tag{134}$$

La dérivée de (134) est discontinue. La préférence est donnée au premier terme, mais ce choix est arbitraire, comme dans le cas de la fonction de Valeur absolue. Encore une fois,

ce choix ne se justifie pas au niveau du formalisme mathématique, mais reste préférable à une dérivation par différences finies.

4.3.2.7 Fonctions Plafond et Plancher

Ces fonctions retournent la partie entière de l'expression qu'elles analysent. La nuance entre les deux est simplement que la fonction Plafond (appelée « ceil » en Tornado) retourne l'entier supérieur alors que la fonction Plancher (appelée « floor ») retourne le plus petit entier et tronque les décimales.

Ainsi, toute fonction passée en argument aux fonctions Plancher ou Plafond sera représentée par un escalier et des sauts discontinus lorsque la valeur numérique passe un nombre entier. Il s'agit donc de dériver une suite de constantes avec une discontinuité entre chaque saut.

Les mathématiques différentielles possèdent une fonction analytique permettant de représenter ces discontinuités. Il s'agit de la fonction Dirac. Cette fonction est une impulsion dont l'intégrale est 1 et dont la largeur tend vers zéro. Sa dérivée est appelée la fonction de Heavyside et est un échelon passant de 0 à 1. Ces deux fonctions permettraient donc de dériver algébriquement toute fonction présentant une discontinuité. Cependant, la fonction Dirac n'est pas implémentée sur la plateforme Tornado. De plus, numériquement, une telle fonction ne peut l'être puisque sa valeur devrait être nulle pour tout x sauf $x = 0$ où sa valeur devrait atteindre l'infini.

Il a donc été décidé de ne dériver que les sections constantes des équations Plafond et Plancher. Comme ces expressions sont constantes sur toute leur plage de variation sauf aux discontinuités, leur dérivée est nulle et les impulsions sont négligées.

$$y = ceil(u) \tag{135}$$

$$\frac{d\big(ceil(u)\big)}{dx} = \frac{d\big(floor(u)\big)}{dx} = 0 \tag{136}$$

4.3.2.8 Fonctions ASSERT, TERMINATE, SIGN, REF et REINIT

Ces fonctions sont par nature non-différentiables puisqu'elles ne retournent pas de nombre réel. En effet, la fonction ASSERT sert à confirmer une condition. Si la condition est fausse, le modèle est arrêté.

La fonction TERMINATE sert à interrompre le modèle en cours de route. La tâche d'interrompre le modèle ne peut revenir au calcul du Jacobien.

La fonction SIGN retourne -1, 0 ou 1 selon le signe de son argument. Comme dans le cas des fonctions Plafond et Plancher, la dérivée de cette fonction est nulle sur tout le domaine d'utilisation sauf en zéro où elle est discontinue.

La fonction REF retourne une référence à une variable (un pointeur en C et C++). Une référence ne peut être dérivée.

La fonction REINIT sert à réinitialiser une variable x avec une variable y. Par sa nature même, cette fonction est systématiquement discontinue sauf dans le cas où $x = y$.

La dérivée de ces cinq fonctions est posée à zéro. Il s'agit d'un choix arbitraire qui pourra être modifié dans le futur. Néanmoins, il est estimé que les erreurs potentielles sont minimes.

4.3.3 Évaluation et discussion sur la dérivation symbolique

Si la dérivation de petits modèles ne pose aucun problème et est exécutée rapidement, dans sa forme actuelle, la dérivation de modèles de la complexité de BSM1 demande des ressources beaucoup trop importantes. Sur un ordinateur équipé d'un processeur Intel Core Duo 2 à 2.0 GHz, il a fallu six heures de calcul pour compiler le modèle et générer le Jacobien (contre quelques secondes pour compiler le modèle seul). Cependant, dans sa forme actuelle, la fonction réalisant la dérivation symbolique n'a servi qu'à démontrer sa capacité à dériver des équations en série et plusieurs améliorations algorithmiques sont possibles et nécessaires. Parmi celles-ci, la plus notable est la gestion des paramètres nuls. En effet, les modèles de station d'épuration sont générés automatiquement et de très nombreuses opérations sont réalisées bien qu'elles n'aient aucune utilité. Ce fait découle

directement de la construction de modèle sur la matrice de Gujer (aussi appelée matrice de Petersen).

La matrice de Gujer représente un modèle en ayant une variable d'état par colonne et un processus de transformation et sa cinétique par rangée (Henze et al. 2000). Les éléments de la matrice sont alors les termes stœchiométriques de production ou de consommation de la variable d'état. Il est ainsi possible de décrire à l'aide d'une seule matrice un modèle entier. Pour construire les équations différentielles de ce modèle, il suffit alors de faire la somme pour chaque variable d'état du produit du terme stœchiométrique et de sa cinétique. Puisque les processus vont de la croissance de la biomasse hétérotrophe à la nitrification de l'azote organique, il est évident que toutes les variables d'état ne sont pas affectées par tous les processus ayant cours dans le modèle. Cependant, les modèles générés automatiquement par WEST et utilisés sur la plateforme Tornado sont construits à l'aide d'une somme de produits où tous les procédés apparaissent dans toutes les équations d'état. Les procédés ne contribuant pas aux différentes variables d'état sont simplement gérés par leur coefficient stœchiométrique qui est nul.

La conséquence première de cette construction automatique de modèles est un nombre effarant de paramètres dans le modèle ayant une valeur nulle. Ainsi, lors des premières dérivations symboliques du Jacobien du modèle ASM1 – Décanteur ponctuel, plusieurs éléments nuls du Jacobien se voyaient attribuer une expression algébrique dont le résultat était systématiquement zéro. C'est pourquoi, à l'heure actuelle, chaque paramètre inséré dans une équation du Jacobien est testé pour garantir qu'il est non-nul. Évidemment, énormément de ressources sont investies dans ces vérifications. Une première étape nécessaire sera donc de nettoyer les modèles idéalement avant même la compilation, mais à tout le moins avant de lancer la dérivation du Jacobien des modèles.

Une autre direction de recherche prometteuse pour accélérer tant la dérivation que le calcul en simulation du Jacobien est la création d'équations intermédiaires. Présentement, lorsqu'une variable est rencontrée, l'équation correspondante est recherchée dans la liste des équations pour être dérivée et être incluse dans la dérivée. Compte tenu de la grande complexité des modèles, des dérivées comptant plus de 600 termes apparaissent dans le Jacobien du modèle ASM1 – Décanteur Ponctuel. Pour un modèle plus complexe comme le

modèle BSM1, une équation de 3300 termes apparaît. Puisque des équations intermédiaires peuvent parfois être réutilisées dans d'autres dérivées, leur extraction d'une longue équation permettrait d'abord de raccourcir ces équations et d'accélérer le calcul du Jacobien si certaines d'entre elles sont réutilisées dans d'autres équations.

5 Conclusion

5.1 Objectifs de l'étude

L'étude présentait deux objectifs principaux, soit l'accélération du calcul du régime dynamique d'une simulation et l'amélioration du calcul du régime permanent.

L'accélération du calcul d'une simulation utilise une simplification du modèle en séparant les équations d'état en deux catégories, soit les équations dont la constante de temps est très courte et les équations présentant une constante de temps plus longue. La section rapide (constante de temps courte) est considérée comme à l'équilibre en tout temps, c'est-à-dire que sa dérivée est conservée à zéro. La section lente est, quant à elle, résolue par un intégrateur classique en régime dynamique. Ainsi, le régime transitoire des variables rapides étant d'un intérêt très limité (par exemple, variations à la seconde de la concentration en oxygène dissous dans l'opération sur une semaine d'une station d'épuration), il est négligé et seule sa valeur à l'équilibre est considérée. Il est donc possible de construire la solution de la simulation en utilisant un pas de temps moyen supérieur à celui nécessaire pour solutionner le modèle original. Le temps nécessaire pour simuler le modèle en entier peut donc s'en trouver réduit. Pour implémenter une telle simplification, un algorithme de type Diagonal Implicite de Runge-Kutta (DIRK) a été implémenté. Les avantages de ce type d'intégrateur sont multiples. Il faut noter, entre autre, sa très grande stabilité lui permettant de gérer des modèles raides et sa capacité à résoudre des systèmes d'équations de type ÉDA plutôt qu'uniquement des équations différentielles ordinaires. Ainsi, les équations rapides d'un modèle simplifié peuvent être résolues comme des équations algébriques implicites non-linéaires, c'est-à-dire en posant leur dérivée à zéro.

Puisqu'un algorithme supplémentaire est implémenté (DIRK), ce dernier doit également prouver qu'il est capable de performances au moins similaires ou idéalement supérieures aux algorithmes performants équivalents. Puisque chaque algorithme présente des forces et des faiblesses, ces caractéristiques doivent être relevées.

Le calcul du régime permanent est un cas particulier d'une simulation dynamique qui est atteint lorsque toutes les variables d'état sont à l'équilibre. Mathématiquement, cet état est reconnu lorsque les dérivées sont nulles. Un algorithme spécifique au calcul du régime permanent consiste donc à déterminer les valeurs des variables d'état annulant toutes les équations différentielles. Puisque le régime transitoire du modèle n'est pas recherché, les algorithmes ne cherchent qu'à minimiser les dérivées, ne portant aucune attention aux valeurs finales vers lesquelles la solution se dirige. La conclusion fréquente est que la solution détectée, acceptable d'un point de vue mathématique (les équations différentielles sont annulées), ne l'est pas d'un point de vue physique. Ainsi, il est fréquent de voir des masses ou concentrations finales négatives. De plus, les algorithmes spécifiques étant très spécialisés, ils supportent souvent très mal les non-linéarités inhérentes aux modèles et échouent à trouver une solution. Dans le but de favoriser la convergence des algorithmes, cette étude souhaite utiliser un maximum d'information venant de l'utilisateur de l'algorithme. Cette information vient généralement sous la forme d'estimés initiaux des variables d'état, soit une première approximation des valeurs finales. Cependant, d'autres informations, comme la plage sur laquelle devrait se retrouver les variables d'état peuvent également être introduites à l'algorithme.

Finalement, des ressources importantes doivent être investies dans le calcul du Jacobien d'un modèle. Ce Jacobien, la matrice des dérivées partielles des fonctions d'état par rapport aux variables d'état, est utilisé intensivement dans les algorithmes de calcul du régime permanent de même que dans de nombreux intégrateurs. Il est traditionnellement calculé numériquement par différences finies, ce qui nécessite de nombreuses évaluations du modèle et qui peut générer des baisses significatives de performance des algorithmes. Ainsi, des techniques pour réaliser une dérivation symbolique, donc exacte, du Jacobien sont développées.

5.2 Résumé de l'étude

5.2.1 Régime dynamique et simplification de modèle

La simplification automatique du modèle a été réalisée sur la base d'un algorithme Diagonal Implicite de Runge-Kutta. L'implémentation de base et les paramètres de la méthode utilisée sont tirés des travaux de (Cameron 1983). Cet algorithme est intéressant pour sa très grande stabilité sur les modèles raides et pour sa capacité à solutionner des modèles composés d'équations différentielles et algébriques implicites alors que la majorité des intégrateurs se contente habituellement à résoudre des équations différentielles ordinaires uniquement. Cette capacité supplémentaire est utilisée dans la simplification des modèles en déplaçant une section du modèle de la section des équations différentielles à la section des équations algébriques. Ces équations sont ensuite résolues à chaque pas de temps en calculant leur valeur annulant leur dérivée dans le temps, c'est-à-dire en les ramenant à l'équilibre.

Cependant, il n'est pas possible de déterminer facilement d'un point de vue purement numérique quelles variables d'état méritent le titre de variables rapides. La séparation du modèle est donc réalisée sur la base de l'analyse des valeurs propres du Jacobien du modèle. Puisqu'une valeur propre est liée à la constante de temps d'une variable, il est possible de trier les variables d'état en fonction de leurs constantes de temps, donc de leur valeur propres. Le problème restant est donc de lier variable d'état et valeur propre puisque le calcul de ces dernières renvoie une liste triée de valeurs. C'est ici que la technique d'homotopie développée par (Steffens et al. 1997) est intégrée. Les valeurs propres d'un Jacobien se limitant à la diagonale de la matrice originale peuvent être liées facilement aux variables d'état. Par la suite, une contribution du Jacobien original de plus en plus grande permet de suivre le lien entre valeur propre et variable d'état. Le modèle possède alors un indicateur de la vitesse de changement de chaque variable d'état, permettant de les trier.

L'un des inconvénients de la méthode proposée par (Steffens et al. 1997) est que l'analyse est réalisée une seule fois avant la simulation et utilise les variables d'état au régime permanent. Puisque les conditions peuvent changer significativement durant le régime

dynamique, cette analyse à priori peut être fausse. Ce travail propose donc d'appliquer la technique d'homotopie durant la simulation du modèle. Ainsi, à intervalles réguliers définis soit par le temps simulé écoulé, soit par le nombre de pas parcouru, le Jacobien est réévalué et l'analyse de ses valeurs propres à l'aide de la technique d'homotopie est réalisée pour confirmer ou infirmer les simplifications du modèle.

Il n'est cependant pas suffisant de réévaluer la simplification du modèle à intervalle régulier. Des mécanismes ont également été mis en place pour confirmer ou infirmer la validité du modèle simplifié au cours de l'évolution de la simulation. Ainsi, deux mécanismes sont utilisés. Le premier s'assure qu'après une réduction, le nombre de variables d'état rapides n'ait pas diminué. Dans le cas où ce nombre diminue, une ou plusieurs variables d'état ne peuvent être résolues par le modèle simplifié puisque leur régime dynamique est d'intérêt et est éliminé par la simplification. Le second mécanisme confirme qu'aucune variable d'état n'ait subie un changement de signe sous le modèle simplifié. L'hypothèse à la base de ce test est qu'en traitement des eaux, les variables d'état sont généralement des masses ou des concentrations. Une variable d'état avec une valeur négative est donc hautement suspecte. Pour éviter que cette valeur négative soit un artéfact de calcul dû à une mauvaise solution numérique, tout changement de signe dans une variable doit être calculé à partir du modèle original. Ainsi, un test peu coûteux en ressources permet de détecter toute valeur suspecte.

5.2.2 Régime permanent et ajout de limites

Le calcul du régime permanent à l'aide d'algorithmes standards peut mener à un ensemble de solutions dont la plupart n'ont aucune signification physique (masse négative, par exemple). De plus, l'utilisateur d'un tel algorithme a généralement une bonne idée des valeurs minimales et maximales que peuvent prendre les variables d'état à l'équilibre. Des limites ont donc été ajoutées pour diriger la convergence dans la zone d'intérêt pour chaque variable d'état. Ces limites prennent la forme d'équations algébriques implicites dont la solution dépend de la variable d'état à contraindre et d'une variable fictive associée. Ces équations ne peuvent alors être résolues que si la variable d'état est à l'intérieur des limites qui lui sont imposées.

Les équations de limite proposées peuvent contraindre une variable d'état à être supérieure à un minimum, inférieure à un maximum ou encore comprise entre un minimum et un maximum, chaque cas présentant une équation standard. Les résultats ont démontré que de telles limites interdisent efficacement les solutions hors des contraintes et élargissent dans certains cas la zone d'estimés initiaux convergeant vers la solution.

5.2.3 Dérivation symbolique du Jacobien

Le Jacobien est l'outil mathématique le plus utilisé dans le cadre de ce mémoire. Cependant, il est toujours évalué numériquement. Une fonction a donc été développée sur la plateforme Tornado pour dériver tous les termes du Jacobien au moment de la compilation du modèle. Si les temps de calcul sont prohibitifs à ce stade, il a tout de même été démontré qu'il était possible de dériver ces équations malgré la présence de termes non-analytiques comme des instructions conditionnelles (IF – ELSE).

5.3 Principales Conclusions

5.3.1 Régime dynamique

L'algorithme DIRK développé a démontré d'excellentes performances à faible précision, ne démontrant aucune dérive significative des variables d'état. Si des calculs ne nécessitant pas de précision importante sont suffisants, l'utilisation de cet algorithme devrait être préférée à l'utilisation de l'algorithme CVODE. En effet, ce concurrent direct et plus performant à forte précision a laissé voir une dérive importante (de l'ordre de quelques points de pourcentage) de certaines variables d'état. Malheureusement, si les performances de l'algorithme DIRK à faible précision sont impressionnantes, il n'en va pas de même à forte précision où les temps de calcul ne sont plus compétitifs avec ceux des algorithmes concurrents comme ce même CVODE.

La simplification automatique de modèle n'a pas encore rempli toutes ses promesses, non plus. En effet, le but premier d'un modèle simplifié est d'être plus simple à résoudre. Les temps de calcul en jeux prouvent le contraire et montrent qu'il est plus rapide de

solutionner le modèle original que le modèle simplifié. Deux éléments sont pointés du doigt pour expliquer ces performances décevantes :

1. La nécessité de résoudre un ensemble d'équations algébriques implicites à chaque pas de l'algorithme

L'algorithme DIRK possède la capacité de résoudre des systèmes d'équations différentielles et algébriques (ÉDA) de même que des systèmes d'équations différentielles ordinaires (ÉDO). Malheureusement, le temps nécessaire à la solution de ces deux types d'équations n'est pas équivalent. La solution d'équations implicites algébriques non-linéaires nécessite une utilisation supplémentaire de la méthode de Newton-Raphson à chaque pas. Bien que l'algorithme DIRK fournisse de bons estimés initiaux à chaque pas, ce surcoût n'est pas négligeable et augmente en fonction du nombre d'équations à résoudre.

2. La nécessité de contrôler la solution à chaque pas du modèle simplifié

Le modèle simplifié se rapproche du modèle original, mais peut tout de même présenter un comportement complètement différent. Le contrôle de sa solution requière d'abord un plus grand nombre d'évaluations du Jacobien et de ses valeurs propres. Bien que consommant moins de ressources que la résolution des équations algébriques, il s'agit d'une demande de calcul non négligeable.

5.3.2 Régime permanent

La principale conséquence de l'utilisation de limites est bien sûr l'élimination de toutes les solutions hors des contraintes imposées. Bien que la taille du modèle augmente significativement (allant jusqu'à doubler le nombre de variables d'état) et donc le temps de calcul pour trouver la solution, le surcoût de l'utilisation de limites peut être justifié par cette disparition des solutions indésirables. En plus, dans certains cas comme celui du modèle Prédateur – Proie, la zone d'estimés initiaux convergeant à la solution désirée augmente, ce qui permet une plus grande flexibilité à l'utilisateur. Cependant, des résultats comme ceux du modèle ASM1 – Décanteur ponctuel montrent une zone de convergence mal définie où une variation mineure sur les estimés initiaux mène à un échec de l'algorithme. Les stratégies de recherche devraient donc offrir la possibilité de donner un

estimé initial à chaque variable d'état et également une zone de recherche en cas d'échec. Compte tenu du coût très faible en temps de calcul de la recherche du régime permanent contrairement à l'exécution d'une simulation dynamique complète, le surcoût associé à de nombreux essais est négligeable si ces essais mènent à la solution.

5.3.3 Dérivation symbolique du Jacobien

Bien qu'il soit possible de dériver symboliquement un modèle, il reste beaucoup de travail à réaliser sur cette fonction. En effet, les temps de calcul sont présentement prohibitifs, nécessitant des heures de calcul pour un modèle pouvant être compilé en moins d'une minute. Cependant, le simple fait d'éliminer les paramètres nuls d'un modèle devrait diminuer significativement le temps de calcul puisque l'algorithme de dérivation n'irait plus se perdre dans des branches inutiles des équations d'état.

La dérivation symbolique du Jacobien d'un modèle devrait montrer toute sa puissance sur de grands modèles de station d'épuration. Ainsi, tel que l'a démontré le Jacobien du modèle BSM1, alors qu'un modèle ASM1 est très couplé, c'est-à-dire que chaque variable dépendra plus ou moins de l'ensemble des variables, un modèle de haut niveau composé d'un nombre important de sous-modèles sera fortement découplé puisque chaque sous-modèle n'interagira avec les autres qu'au moyen d'une ou deux variables de transport. La conséquence directe est que le Jacobien du modèle BSM1 ne comporte qu'environ 10% d'éléments non-nuls. Une dérivation numérique doit évaluer chaque terme peu importe le nombre d'éléments nuls, alors qu'une dérivation symbolique n'évaluera que les termes variables, sauvant du coup un nombre potentiellement élevé de calculs.

De plus, la forme particulière du Jacobien suggère que plusieurs variables d'état sont totalement indépendantes les unes des autres. Une telle information peut également être utilisée pour accélérer le calcul d'un Jacobien numérique en perturbant plusieurs variables d'état indépendantes simultanément.

5.4 Travaux futurs

5.4.1 Algorithme DIRK

L'algorithme DIRK utilise actuellement une décomposition LU pour résoudre ses opérations matricielles sur le Jacobien. Puisque ce Jacobien doit être remis à jour périodiquement avec des changements mineurs, une méthode comme la décomposition QR devrait permettre des gains de calcul important en permettant une mise à jour peu coûteuse du Jacobien. Les gains d'une telle méthode sont doubles puisque présentement, le Jacobien est évalué une fois et est utilisé tant qu'il est jugé à jour, pouvant donc calculer quelques pas sans avoir à être réévalué. La différence entre ce Jacobien et l'état réel du modèle peut être la cause d'une ou deux itérations supplémentaires à chaque étape de la méthode de Runge-Kutta. Si un pas nécessite quatre étapes, il s'agit donc de près d'une dizaine d'itérations supplémentaires pour calculer ce pas par rapport à l'utilisation d'un Jacobien à jour. Cependant, comme un Jacobien requière n+1 évaluations du modèle, le coût supplémentaire d'un Jacobien « âgé » est justifié. Cependant, une décomposition QR permet de mettre à jour la décomposition d'un Jacobien sur la base de la dernière évaluation du modèle à condition que le nouveau soit assez proche de l'ancien. De plus, avantage additionnel, ce n'est pas le Jacobien comme tel qui est mis à jour, mais directement sa décomposition, sauvant une étape supplémentaire.

Il est donc possible d'espérer des gains de performance significatifs si une décomposition QR et sa mise à jour sont implémentés sur l'algorithme DIRK plutôt qu'une décomposition LU.

Que l'usage d'une décomposition QR soit possible ou non, le Jacobien reste la pierre angulaire de l'algorithme DIRK. Ce faisant, il demande également la part du lion des ressources que l'algorithme consomme. Puisque les modèles de stations d'épuration sont appelés à se complexifier, travailler avec un Jacobien entier deviendra de plus en plus demandant puisque sa taille est le carré du nombre de variables d'état. Des recherches du côté de l'utilisation de matrices creuses et de la simplification du Jacobien pourraient donc permettre des gains en performance.

L'algorithme peut actuellement être utilisé pour résoudre des ensembles d'équations différentielles ordinaires ou des ensembles d'équations différentielles et algébriques. Cependant, le code de l'algorithme demeure lourd et laisse penser qu'une optimisation des fonctions est toujours possible. Ainsi, une méthode proposée par (Cameron 1983) devrait permettre de résoudre les système d'équations différentielles et algébriques sans surcoût pour le système d'équations algébriques. La méthode requière simplement une modification des paramètres de l'algorithme pour que les résultats intermédiaires des équations algébriques de l'algorithme soient identiques aux résultats finaux. Malheureusement, cette méthode n'a pu être implémentée principalement par manque de temps. Il est pourtant possible d'espérer une réduction significative du temps de calcul.

5.4.2 Réduction de modèle

Les performances actuelles en termes de temps de calcul des modèles simplifiés sont moins bonnes que celles des modèles originaux. Parmi les causes possibles de ces résultats, il y a bien le calcul des équations algébriques à chaque pas. Cependant, il serait nécessaire de déterminer quels modèles pourraient profiter d'une simplification. En effet, si une résolution d'un pas aux 90 secondes est voulue et que le modèle original renvoie une résolution d'un pas aux 13 secondes, l'usage de la réduction peut être superflu. Des lignes directrices quant à l'usage d'un modèle réduit seraient donc souhaitables pour aider à l'utiliser dans les contextes appropriés.

Dans le cas d'un modèle requérant une simplification, l'algorithme DIRK pourrait également ne pas être l'algorithme le plus approprié pour résoudre le modèle simplifié. En effet, puisque l'algorithme est infiniment stable sur des modèles linéaires et beaucoup plus stable que les intégrateurs explicites en générale sur les modèles non-linéaires, la longueur du pas est généralement contrainte par la précision de l'algorithme plutôt que par la stabilité. Ainsi, ses performances sur les modèles raides sont limitées par les mêmes éléments que les performances d'un modèle réduit similaire (entrées dynamiques, contrôleurs, discontinuités, etc.). Cependant, la combinaison de différents algorithmes pourrait être envisagée. Ainsi, les équations rapides qui déstabilisent un algorithme comme Runge-Kutta 4 (RK4), pourraient être résolues par une méthode implicite comme DIRK

tandis que les équations lentes seraient solutionnées par RK4. Étant explicite, la méthode RK4 n'est qu'une somme pondérée de quatre évaluations des fonctions d'état. Si la stabilité n'est pas un problème, cet algorithme demeure la technique la plus puissante disponible en termes de performances. L'intérêt d'une telle méthode est de réduire de façon considérable la taille des matrices sur lesquelles l'algorithme DIRK doit travailler. En effet, le modèle BSM1 voit 30 variables d'état être considérées comme rapides. Si seules ces 30 variables sont considérées pour la construction du Jacobien, la taille de la matrice passe de 108×108 à 30×30, soit une réduction de 72% du nombre de variables et de 92% du nombre d'éléments de la matrice. Les itérations sont donc beaucoup moins coûteuses à réaliser pour le calcul de la solution du pas.

5.4.3 Lien entre valeurs propres et variables d'état

La méthode d'Homotopie permet de lier efficacement les valeurs propres du Jacobien aux variables d'état. Cependant, dans l'implémentation actuelle de l'algorithme DIRK, le lien entre la diagonale du Jacobien et le Jacobien entier est réalisé à l'aide de vingt étapes intermédiaires. Tout comme la décomposition d'une matrice sous forme LU, le temps de calcul des valeurs propres de cette matrice augmente proportionnellement au carré du nombre de variables d'état. Pour les très grandes matrices, ces temps de calculs en viennent à prendre une part significative du temps total de calcul de la simulation. L'usage de routines numériques standard oblige l'utilisation de cette technique d'homotopie. Cependant, le développement de routines maisons pour calculer les valeurs propres ou une approximation acceptable pourrait permettre de calculer ces dernières et de les lier directement aux variables d'état. Aucune recherche n'a été réalisée en ce sens dans le cadre des travaux présents, mais la simplification des modèles aurait beaucoup à gagner à se libérer de la technique d'homotopie utilisée.

Dans l'hypothèse où la méthode d'homotopie est la seule acceptable, il serait nécessaire d'optimiser le nombre d'évaluations des valeurs propres sur des matrices intermédiaires. De plus, actuellement, la technique d'homotopie est utilisée systématiquement pour séparer le modèle. Cependant, les résultats antérieurs sont perdus. Il y aurait cependant la possibilité de suivre l'évolution des valeurs propres, par exemple, à chaque évaluation du

Jacobien. Puisque les variations y seraient mineures, les liens devraient être simples à maintenir.

5.4.4 Calcul du régime permanent

L'un des objectifs du présent travail était de développer un algorithme performant pour calculer le régime permanent d'un modèle. Dans le cadre de modèles biologiques de stations d'épuration, cette performance a été étudiée, non pas sous l'angle du temps de calcul, mais sous celui de n'offrir que des solutions acceptables. L'ajout de limites aux variables d'état permet d'éliminer les résultats indésirables. Le problème le plus important est désormais un échec de l'algorithme. L'expérience montre que ces échecs peuvent être dus à deux causes principales, soit un estimé initial loin de la solution ou une difficulté numérique ponctuelle.

Ainsi, si l'utilisateur a une bonne idée de la zone dans laquelle peut se retrouver le régime permanent, cette information peut également être utilisée pour générer des estimés initiaux. Une stratégie de détermination d'estimés initiaux représente donc l'étape suivante logique dans le calcul du régime permanent et devrait permettre de solutionner la majorité des modèles présentant des difficultés. Une telle stratégie pourrait, par exemple, être de fournir des bornes à chaque variable d'état et choisir les estimés initiaux sur la base d'une méthode de Monte-Carlo jusqu'à ce qu'une solution soit trouvée. Compte-tenu de la rapidité des algorithmes de recherche du régime permanent par rapport à l'exécution d'une simulation dynamique, le fait de devoir lancer plusieurs recherches devrait toujours être plus performant que la solution longue (la simulation dynamique).

5.4.5 Dérivation symbolique

Les travaux menés sur la dérivation symbolique du Jacobien d'un modèle complexe de station d'épuration sont au niveau exploratoire. Ainsi, plusieurs améliorations sont encore nécessaires avant de pouvoir utiliser toute la puissance d'un Jacobien symbolique.

Un modèle comportant n variables se voit actuellement dérivé en n^2 équations autonomes. Puisqu'une même expression mathématique pourrait avoir à être dérivée par rapport à une même variable à répétition, une procédure d'écriture des équations dérivées intermédiaires

doit encore être écrite. Sur des modèles complexes, cela permettrait de conserver un nombre raisonnable de variables par équation tout en réduisant les temps de calcul dans le modèle. Des techniques de simplification et/ou optimisations d'équations seraient également intéressantes.

En termes de performances, le principal frein actuel semble être la gestion des paramètres nuls. Ces paramètres proviennent de modèles générés automatiquement. Un exemple typique est l'équation du volume de liquide dans un réacteur où chaque composant du modèle se voit attribuer une masse volumique. Cependant, comme seul le volume de l'eau est considéré, les paramètres des autres variables d'état sont tous posés à zéro. Ainsi, dans un modèle comme le modèle ASM1 – Décanteur ponctuel, pour un paramètre non-nul, 13 paramètres nuls apparaissent. Dans le cadre du calcul du modèle, il s'agit d'une surcharge assez faible. Mais lorsque vient le temps de dériver ces expressions, le nombre d'expression a explosé et les temps de calcul aussi. Une fonction de nettoyage du modèle est donc jugée nécessaire et devrait également profiter à l'exécution des modèles en simulation.

L'étude du Jacobien du modèle BSM1 ouvre la porte à une évaluation hybride du Jacobien. En effet, la dérivation symbolique permet d'estimer le coût en temps de calcul de chaque élément du Jacobien en comptant le nombre d'opérations mathématiques nécessaire pour la solution. Puisqu'un Jacobien numérique est actuellement calculé colonne par colonne, si le coût de tous les éléments d'une colonne est inférieur au coût de l'évaluation du modèle, cette colonne peut être évaluée symboliquement. Dans le cas contraire, une évaluation numérique sera préférée.

Actuellement, les fonctions non-analytiques (instruction conditionnelle `IF-ELSE`, fonction `PREVIOUS`, etc.) représentent un défi de la dérivation symbolique. En effet, les dérivées proposées sont justifiables lorsque comparées à la solution exacte, mais leur mathématique formelle n'est pas garantie. Ces fonctions gagneraient donc à avoir une justification mathématique.

6 Bibliographie

Alex, J. (2008). Communication personnelle. Mont-St-Anne, Canada.

Alexander, R. (2003). "Design and implementation of DIRK integrators for stiff systems." Applied Numerical Mathematics **46**(1): 1-17.

Batstone, D. J., J. Keller, R. I. Angelidaki, S. V. Kalyuzhnyi, S. G. Pavlostathis, A. Rozzi, W. T. M. Sanders, H. Siegrist and V. A. Vavilin (2002). Anaerobic Digestion Model No.1 IWA Publishing, London, UK.

Broyden, C. G. (1965). "A class of methods for solving nonlinear simultaneous equations." Mathematics of Computation **19**(92): 577-593.

Bui, T. D. (1979). "Some A-stable and L-stable methods for the numerical integration of stiff ordinary differential equations." Journal of the ACM **26**(3): 483-493.

Cameron, I. T. (1983). "Solution of differential-algebraic systems using diagonally implicit Runge-Kutta methods." IMA Journal of Numerical Analysis **3**: 273-289.

Cameron, I. T. (2008). Communication personnelle. Brisbane, Australie.

Cameron, I. T. and R. Gani (1988). "Adaptive Runge-Kutta algorithms for dynamic simulation." Computers & Chemical Engineering **12**(7): 705-717.

Claeys, F. H. A. (2008a). A Generic Software Framework for Modelling and Virtual Eperimentation with Complex Biological Systems, PhD thesis. Ghent University, Belgium, 303 p.

Claeys, F. H. A., P. Fritzson and P. A. Vanrolleghem (2007). "Generating efficient executable models for complex virtual experimentation with the Tornado kernel." Water Science and Technology **56**(6): 65-73.

Claeys, P. (2008b). Towards Automatic Numerical Solver Selection for Hierarchical Bioprocess Models, PhD thesis. Ghent University, Belgium, 285 p.

Copp, J. B., Ed. (2002). The COST Simulation Benchmark - Description and Simulator Manual. Luxembourg, Office for Official Publications of the European Communities.

De Pauw, D. J. W. and P. A. Vanrolleghem (2006). "Practical aspects of sensitivity function approximation for dynamic models." Mathematical and Computer Modelling of Dynamical Systems **12**(5): 395-414.

Dochain, D. and P. A. Vanrolleghem (2001). Dynamical Modelling and Estimation in Wastewater Treatment Processes. IWA Publishing, London, UK.

Edgar, T. F., D. M. Himmelblau and L. S. Lasdon (2001). Optimization of Chemical Processes. McGraw-Hill Chemical Engineering Series McGraw-Hill, New York.

Farrow, L. A. and D. Edelson (1974). "Steady-state approximation - fact or fiction." International Journal of Chemical Kinetics **6**(6): 787-800.

Garneau, C., D. J. Batstone, F. H. A. Claeys and P. A. Vanrolleghem (2009). "Stiffness reduction of complex non-linear models and procedure to maintain solution quality." In: Proceedings of the 18th IMACS World Congress - MODSIM09 International Congress on Modelling and Simulation, Cairns, Australia, 13-17 July 2009.

Gujer, W. (2006). "Activated sludge modelling: past, present and future." Water Science and Technology **53**(3): 111-119.

Hangos, K. M. and I. T. Cameron (2001). Process Modelling and Model Analysis. Process systems engineering series, Academic Press, London, UK.

Henze, M., C. P. L. Grady, W. Gujer, G. v. R. Marais and T. Matsuo (1987). Activated Sludge Model No. 1 IAWQ, London, UK.

Henze, M., W. Gujer, T. Mino and M. van Loosdrecht (2000). Activated Sludge Models ASM1, ASM2, ASM2d and ASM3IWA Publishing, London, UK.

Hesstvedt, E., O. Hov and I. S. A. Isaksen (1978). "Quasi-steady-state approximations in air-pollution modeling - comparison of two numerical schemes for oxidant prediction." International Journal of Chemical Kinetics **10**(9): 971-994.

Kakizaki, M. and T. Sugawara (1985). "A modified Newton method for the steady-state analysis." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **4**(4): 662-667.

Levine, J. R., T. Mason and D. Brown (1992). Lex & yacc. Nutshell handbook O'Reilly & Associates, Inc., Sebastopol, Calif.

Moré, J. J., B. S. Garbow and K. E. Hillstrom (1980). User Guide for MINPACK-1. Argonne National Laboratory Report ANL-80-74 Argonne, Il, USA.

Okino, M. S. and M. L. Mavrovouniotis (1999). "Simplification of chemical reaction systems by time-scale analysis." Chemical Engineering Communications **176**: 115-131.

Paloschi, J. R. (1994). "A hybrid continuation algorithm to solve algebraic nonlinear equations." Computers & Chemical Engineering **18**: S201-S209.

Powell, M. J. D. (1962). "An iterative method for finding stationary values of a function of several variables." The Computer Journal **5**(2): 147-151.

Press, W. H., W. T. Vetterling, S. A. Teukolsky and B. P. Flannery (1994). Numerical Recipes in C : The Art of Scientific Computing Cambridge University Press, Cambridge, New York.

Reichert, P., D. Borchardt, M. Henze, W. Rauch, P. Shanahan, L. Somlyódy and P. A. Vanrolleghem (2001). River Water Quality Model No. 1. IWA Scientific and Technical Report No. 12. IWA Publishing, London, UK.

Rosen, C., D. Vrecko, K. V. Gernaey and U. Jeppsson (2005). "Implementing ADM1 for benchmark simulations in Matlab/Simulink." In: Proceedings of the First

International Workshop on the IWA Anaerobic Digestion Model No. 1, Lyngby, Copenhagen, Denmark, 4-6 September 2005.

Rosen, C., D. Vrecko, K. V. Gernaey, M. N. Pons and U. Jeppsson (2006). "Implementing ADM1 for plant-wide benchmark simulations in Matlab/Simulink." Water Science and Technology **54**(4): 11-19.

Seppelt, R. and O. Richter (2005). ""It was an artefact not the result": A note on systems dynamic model development tools." Environmental Modelling & Software **20**(12): 1543-1548.

Seppelt, R. and O. Richter (2006). "Corrigendum to “It was an artefact not the result: A note on systems dynamic” [Environ. Model. Softw. 20 (2005) 1543–1548]." Environmental Modelling & Software **21**(5): 756-758.

Shiraishi, E., K. Maeda and H. Kurata (2009). "A gradual update method for simulating the steady-state solution of stiff differential equations in metabolic circuits." Bioprocess and Biosystems Engineering **32**(2): 283-288.

Steffens, M. A., P. A. Lant and R. B. Newell (1997). "A systematic approach for reducing complex biological wastewater treatment models." Water Research **31**(3): 590-606.

Takacs, I., G. G. Patry and D. Nolasco (1991). "A dynamic-model of the clarification thickening process." Water Research **25**(10): 1263-1271.

Vanrolleghem, P. A., D. Borchardt, M. Henze, W. Rauch, P. Reichert, P. Shanahan and L. Somlyody (2001). "River water quality model no. 1 (RWQM1): III. Biochemical submodel selection." Water Science and Technology **43**(5): 31-40.

Annexe A : Tutoriel sur les expériences virtuelles de la plateforme Tornado

Le document qui suit est disponible sur le site web de la plateforme Tornado à l'adresse : http://www2.mostforwater.com/Tornado/Private/index.html.

Introduction

This document illustrates the use of the Tornado command-line tools, on the basis of a simple flat Modelica model. Focus is especially on the creation and modification of various types of virtual experiments.

Note:

- This tutorial only describes the most common use of virtual experiments. Many other advanced features remain unexplored in this document.
- Since this is Tutorial 3, basic knowledge of Tornado is assumed. If this is not the case, it is strongly suggested to first go through Tutorial 1.
- All files created during the tutorial are available here.

Experiments built in this tutorial:

- Simulation
- Steady State Analysis
- Objective Evaluation
- Scenario Analysis
- Optimization
- Sensitivity Analysis
- Scenario Optimization
- Monte Carlo
- Monte Carlo Optimization
- Statistical Analysis
- Ensemble Simulation

Prerequisites

In order to be able to complete this tutorial, the following are required:

- Properly installed copy of Tornado (the tutorial was made on the basis of Tornado-0.30, but other versions might also work to some extent)
- Visualization tool for numerical data (such as GNUplot or Excel)
- Text editor (TextPad, NotePad, etc)
- C compiler (such as MSVC or BCC)

A self-extracting version of Tornado is available on the Software section. Installers for TextPad and GNUplot are available on the Technology section.

Model

The model that will be used through this tutorial is a Mass-Spring system slightly modified to be more realistic (and non-linear) with add-ons such as a variable absorption coefficient and a spring getting looser with time. The aim of this tutorial is to use different kinds of virtual experiments to find the period of resonance.

```
fclass MassSpring

  parameter Real K(unit = "N/(m*kg)") = 100;// Refer to (k/M)^(1/2)
  parameter Real S(unit = "d/kg") = 1;      // Spring fatigue
                                            // factor
  parameter Real M(unit = "kg") = 15;       // Mass
  parameter Real P(unit = "s") = 3;         // Excitation Period
  parameter Real A(unit = "N") = 1;         // Excitation Amplitude
  parameter Real C = 1;                     // Absorption
                                            // coefficient
  parameter Real D = 1e-2;                  // Decay constant of
                                            // the Spring

  constant Real PI = 3.141592654;

  Real C_real(min = 0);
  Real x(start = 0, unit = "m");
  Real v(start = 0, unit = "m/s");
  Real a(unit="m/s^2");
  Real f(start = 0, unit = "N");
  Real K_real(unit = "N/(m*kg)")

  output Real x_out(unit = "m");
  output Real f_out(unit = "N");
  output Real v_out(unit = "m/s");
  output Real a_out(unit = "m/s^2");
  output Real C_out;
  output Real K_out;

initial equation

  K_real = K;

equation

  // Excitation function
  f = A * sin(2 * PI *time / P);

  // Absorption function
  C_real = C * (exp(x) + exp(-x));

  // Explicit formulation of a Mass-Spring System
  a = -(C_real * v + K_real * x) / M + (f / M);

  // Differential equations for speed and position
  der(v) = a;
  der(x) = v;
```

```
    // Time decaying spring constant
    der(K_real) = - D * K_real * exp(-(time * S)/M);

    f_out = f;
    x_out = x;
    v_out = v;
    a_out = a;
    C_out = C_real;
    K_out = K_real;

end MassSpring;
```

Save this model into a file named `MassSpring.mof`. To build the model with Tornado, run the following command:

```
mof2t MassSpring.mof
tbuild MassSpring
```

Now, three files have been generated: `MassSpring.c`, `MassSpring.dll` (on win32) and `MassSpring.SymbModel.xml`. The C code is generated by the mof2t compiler and is compiled and linked into the DLL by `tbuild`. The XML file contains meta-information like parameters value, initial conditions, boundaries, etc.

It can be useful to note that if one wants to rerun an experiment with different parameter values or initial conditions, theses values can be modified in the symbolic model permanently (unless mof2t is called again on the model). This is an alternative if rebuilding the model takes a long time.

Simulation

Needed Experiment: None

At this point, everything is in place to build the first virtual experiment, the simulation. To do so, the command which follows will create a default simulation experiment:

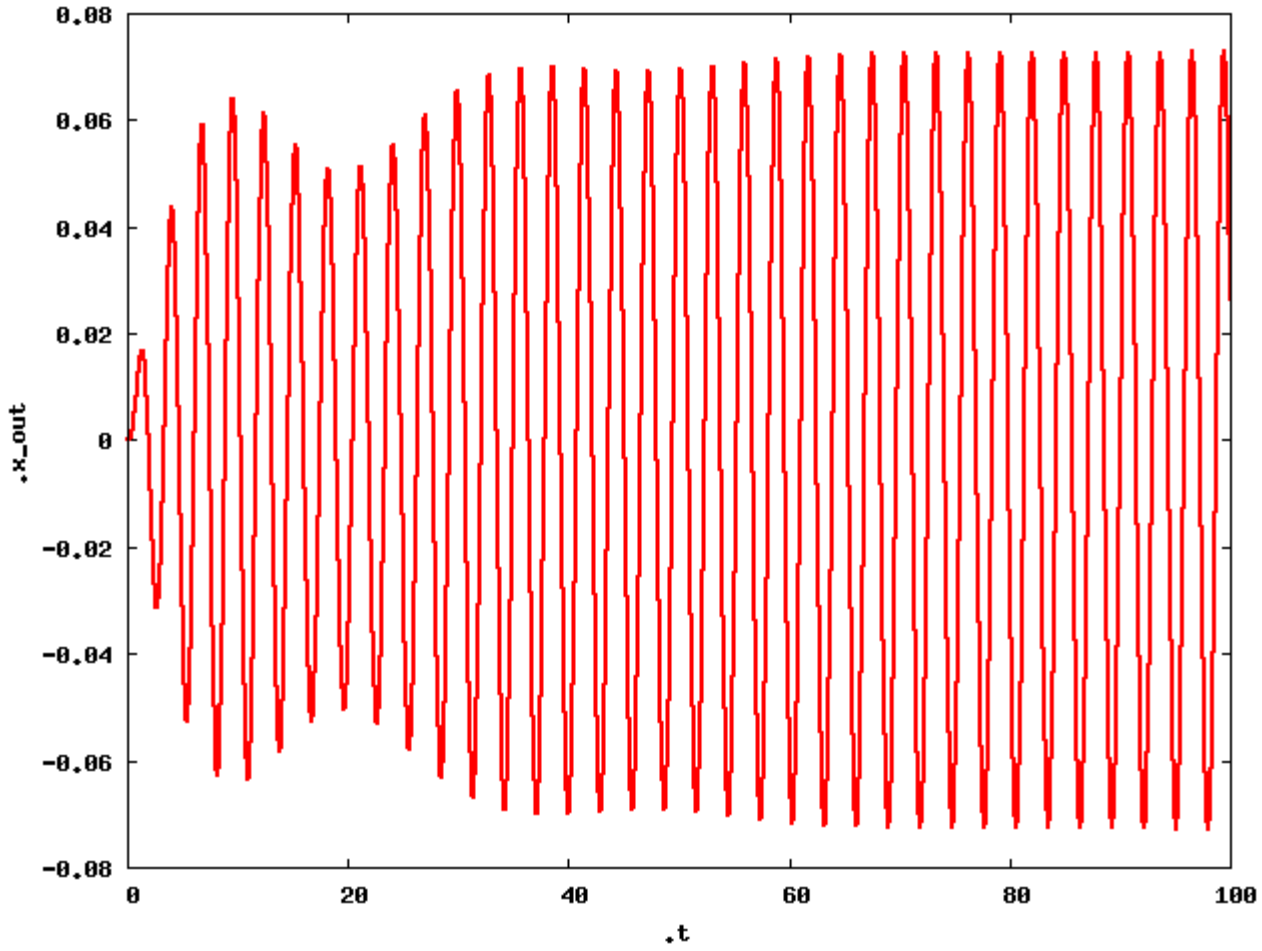
```
tmain ExpCreateSimul MassSpring .
```

In order to have an interesting simulation, it is best to run the experiment from time 0 to 100, since it will show certain phenomena invisible otherwise. To do so, the tag `<Prop Name="StopTime" Value="1"/>` in the newly created experiment must be modified to stop at 100.

The execution of the experiment is triggered by the following command:

```
texec MassSpring.Simul.Exp.xml
```

All results are written in the default output file, unless specified otherwise, `MassSpring.Simul.out.txt`. These results can now easily be visualized through GNUplot or Excel:



Steady State

Needed Experiment: None

As can be imagined, with a steady sinusoidal function, it is impossible to find a steady state. However, in case an experiment is well behaved (i.e. contains a steady state solution) a steady-state experiment can be created for any model through the command:

```
tmain ExpCreateSSRoot <ModelName> .
```

or

```
tmain ExpCreateSSOptim <ModelName> .
```

Where `<ModelName>` stands for the name of the model. The difference between these two experiments is that SSRoot relies on a root finder whereas SSOptim uses an optimization algorithm. In most applications, SSRoot will be faster than SSOptim. These experiments will both return the final values of all derived variables.

Objective Evaluation

Needed Experiment: Simulation

The objective evaluation is the central piece of many virtual experiments. It allows quantifying a set of objectives by returning a single number. If the objective is fully achieved, zero is returned.

So as always, the command line used to create the objective evaluation is:

```
tmain ExpCreateObjEval MassSpring.Simul.Exp.xml .
```

As one can see, the Objective Evaluation experiment stands on a simulation experiment. And what is more, if one opens the newly created experiment, one will see that the simulation experiment has been completely rewritten inside the objective experiment. This means that the Objective Evaluation is independent of the simulation experiment. Since this is often not wanted, it is possible to erase all the simulation information (which is contained between the `<Exp "..." >` and `</Exp>` tags) and to modify `<Exp "..." >` as follows:

```
<Exp Version="1.0" Type="Simul" ...
  FileName="MassSpring.Simul.Exp.xml"/>
```

It is now possible to define the Objective Evaluation. In the scope of a Mass-Spring system, the information of interest will be the resonance of the system. So, one will have a close look at the maximal position the mass will reach. To calculate this maximum automatically, one first needs to add the following tag in the `<Quantities>` section:

```
<Quantity Name=".x_out"/>
```

Many properties can be set with respect to this quantity. However, since it is not possible to remember them all, a tool named `texp` allows to reveal them. So the use of the following command line will rewrite the experiment in a more explicit form:

```
texp MassSpring.ObjEval.Exp.xml
```

Once this is done, it is possible to set the wanted properties to find the maximum value of `.x_out`:

```
<Prop Name="EnableMax" Value="true"/>
<Prop Name="DesiredValueMax" Value="0.5"/>
```

The first line simply tells Tornado to look for the maximum value of the quantity while the second line tells what is the maximum value expected. Finally, if the user had wished to look also for other properties, the Weight options would have allowed to give a different importance to each of the properties, leading to the calculation of a weighted sum of the properties values to define the Objective Evaluation value.

In the case of the mass-spring model, one knows from the first simulation run that the maximum of `.x_out` should never reach 0.5.

The objective evaluation is not meant to be run by itself, although doing so will run the simulation and compute the Objective Value. The usual command to run it is:

```
texec MassSpring.ObjEval.Exp.xml
```

Similarly to a simulation being at the core of the Objective Evaluation, many virtual experiments will need an Objective Evaluation Experiment to work. For instance, the next experiment : the Scenario Analysis.

Scenario Analysis

Needed Experiment: Objective Evaluation

The Scenario Analysis is the simplest experiment that uses the Objective Evaluation. This virtual experiment will allow getting insight in the resonance with respect to the period of excitation. Obviously, the first step is to create the experiment through `tmain`:

```
tmain ExpCreateScen MassSpring.ObjEval.Exp.xml .
```

The experiment settings can now be modified. First, to help maintainability, it may be useful to replace the embedded objective experiment by a link to the one already available. To do so, one has to delete the objective evaluation and to add the following tag:

```
<Exp Version="1.0" Type="ObjEval"
  FileName="MassSpring.ObjEval.Exp.xml"/>
```

To obtain the relation between the maximum displacement of `.x_out` and the period of the excitation, a new tag must be added in the `<Vars>` section:

```
<Var Name=".P"/>
```

Once again, running `texp` will reveal the available options associated with this new tag:

```
texp MassSpring.Scen.Exp.xml
```

Then, the following properties must be set:

```
<Prop Name="RefValue" Value="3"/>
<Prop Name="LowerBound" Value="1"/>
<Prop Name="UpperBound" Value="40"/>
<Prop Name="DistributionMethod" Value="Linear"/>
<Prop Name="NoValues" Value="15"/>
<Prop Name="Spacing" Value="0"/>
<Prop Name="StdDev" Value="0"/>
<Prop Name="UpperBoundPolicy" Value="IncludeUpperBound"/>
<Prop Name="Values" Value=""/>
```

Here, we say that we want 15 simulations using different periods ranging from 1 to 40 with a linear distribution. The default value of the `RefValue` being set at zero, this value must be brought back between Lower and Upper bounds. Although the `RefValue` won't be used through this experiment, to set it at the initial value of P will just serve as a reminder of its former value.

If instead of a fixed number of evaluations, one wanted a fixed spacing between each simulation, then, `NoValue` would be set to zero and `Spacing` to the wanted spacing between each values of `.P`. In this context, the `UpperBoundPolicy` allows to force a last simulation at the upper bound value. Finally, it would also have been possible to specify at which values of `.P` to perform the simulations.

Before executing the experiment, a last option can be switched on. It is the `EnableStorage` which appears in the Objective Evaluation experiment. If its value is set to `true` then each

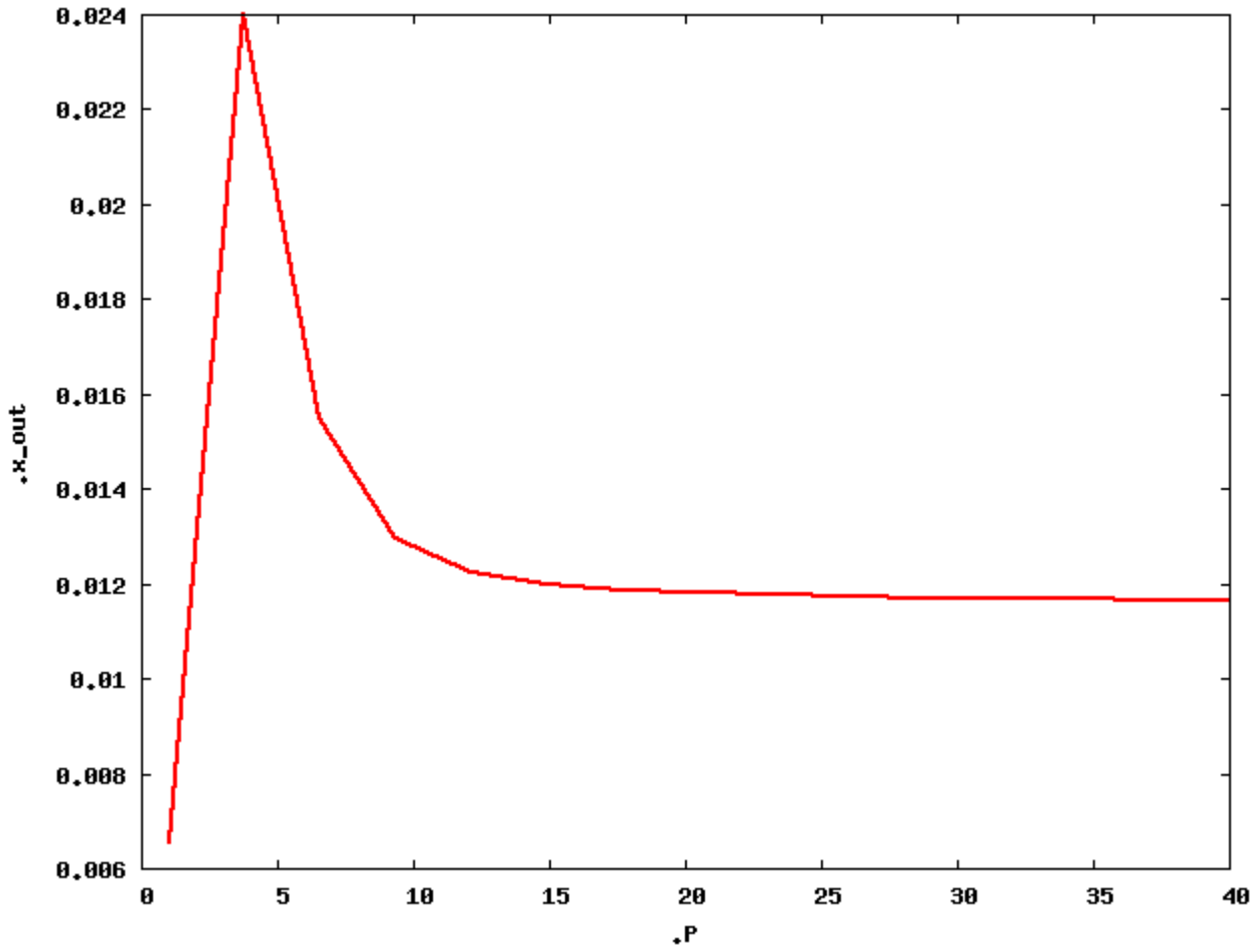
simulation will be recorded under the name `MassSpring.ObjEval.out.txt.{}`, where the `{}` refers to a number from 0 to 14.

The experiment can now be launched with the usual command:

```
texec MassSpring.Scen.Exp.xml
```

If the storage was left to false, only two new text files are generated, instead of seventeen if storage is switched on. The `MassSpring.Scen.log.txt` file records both the properties written in the experiment and the different values used for the parameter `.P`. On the other hand, `MassSpring.Scen.out.txt` contains all information on the objective evaluation for each simulation.

So, at this point, it is possible to plot the value of `.x_out` with respect to `.P`, the period. This figure has been reproduced below:



It is now obvious that the resonance occurs for excitation periods between two and four. It would be possible to refine the scenario analysis by looking for a smaller region and moving always to a better period of resonance. However, such a tool already exists, named the Optimization Experiment.

Optimization

Needed Experiment: Objective Evaluation

The optimization can be used in a way quite similar to the Scenario experiment. The major difference is that values to be tested as `.P` will be chosen automatically with an optimization algorithm. First the Optimization Experiment must be created through the following command:

```
tmain ExpCreateOptim MassSpring.ObjEval.Exp.xml .
```

As before, it is possible to delete the embedded objective evaluation experiment to replace it by the corresponding link:

```
<Exp Version="1.0" Type="ObjEval"
FileName="MassSpring.ObjEval.Exp.xml"/>
```

It is also necessary to add the tag `<Var Name=".P"/>` to the `<Vars>` section. Running `texp` will again state explicitly all available options and properties:

```
texp MassSpring.Optim.Exp.xml
```

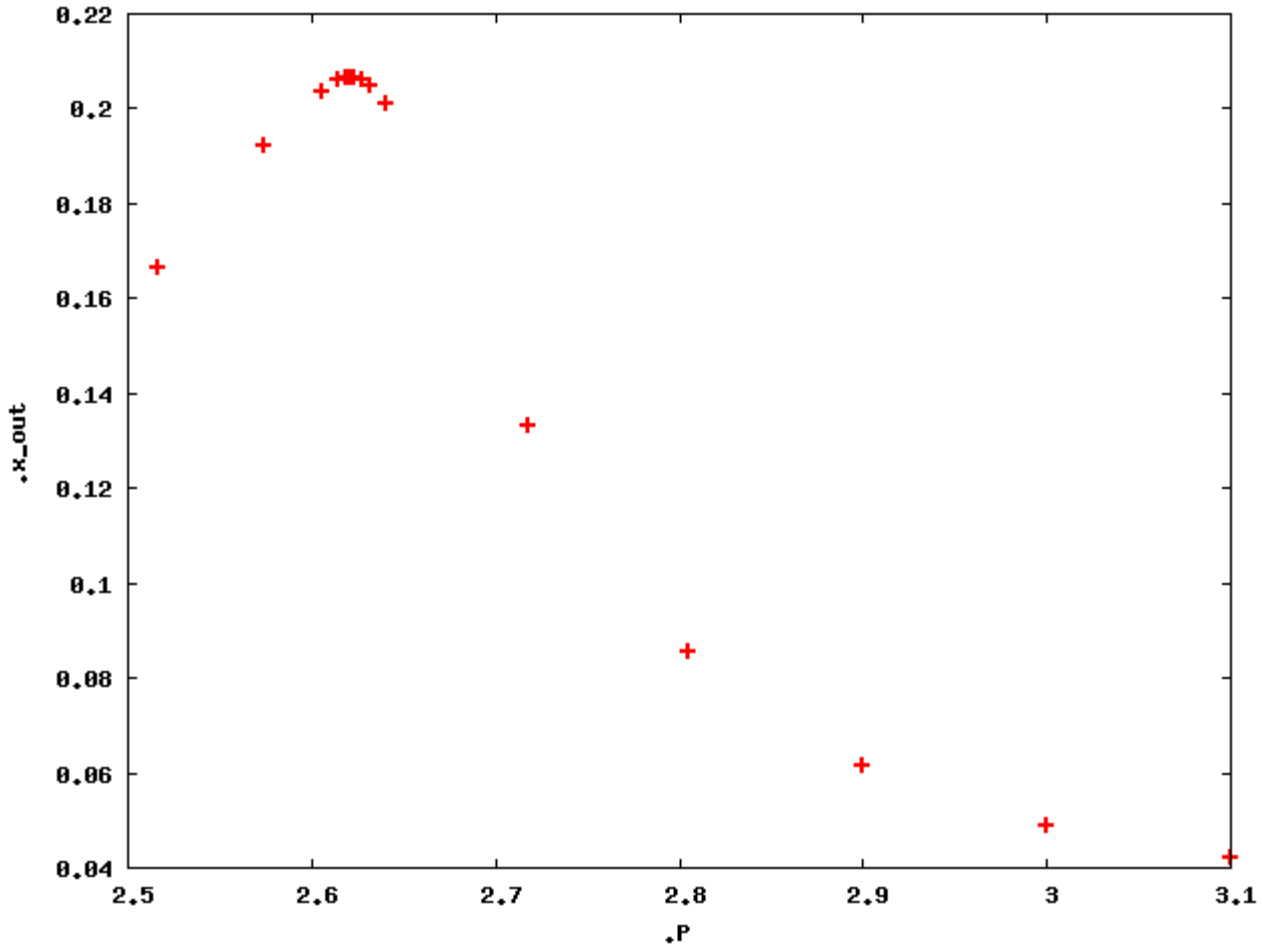
At this point it is possible to use prior knowledge of the model to set properties in an optimal way. The use of the following settings along with the default Optimization algorithm (Simplex) for the variable `.P` will lead to coherent answers:

```
<Prop Name="InitialValue" Value="3"/>
<Prop Name="LowerBound" Value="2"/>
<Prop Name="UpperBound" Value="4"/>
<Prop Name="StepSize" Value="0.1"/>
<Prop Name="Scaling" Value="0.1"/>
<Prop Name="AutoScale" Value="true"/>
```

The experiment can then be run to find where precisely the resonance period is:

```
texec MassSpring.Optim.Exp.xml
```

The sequence of tries made by the experiment can be followed in the file `MassSpring.Otpim.out.txt`. If plotted, the results should look as follow:



To emphasize the importance of prior knowledge of the model, one has to know that the Optimization process consist of minimizing the Objective Value (which is computed through the Objective Evaluation experiment). So, if the initial value of the variable is set carelessly, let's say to 30, with an `UpperBound` set to 40, the Objective Value is almost constant and no optimization can occur. Even if precision if increased, the process is likely to get stuck into a local minimum (which may be caused by numerical error). If this is the case, a number of solutions is available. Among them, there is the change of the algorithm of the experiment. So, by changing `Simplex` or `Praxis`, which are local optimizers, to `GA` or `SA` (two global optimizers) in the experiment, one can hope that a better solution might be found. However, it is obvious that calculation time will be much larger and since the methods rely on tries and error, there is no guarantee that the optimal answer will be found.

Sensitivity Analysis

Needed Experiment: Simulation

Sensitivity analysis is useful to determine whether a small perturbation to a parameter will lead to a small or to a great perturbation of the system. In the case of the Mass-Spring system, the period will be slightly perturbed and the aim is to get an idea on the behaviour of the system. Again, the experiment has to be created.

```
tmain ExpCreateSens MassSpring.Simul.Exp.xml .
```

Once this is done, the quantity under observation must be set in the `<Quantities>` section and the parameter that has to be perturbed in the `<Vars>` section. So the following tags will be inserted in their respective places:

```
<Quantity Name=".x_out"/>
<Var Name=".P"/>
```

Then, in order to see the new sections, `texp` must be called.

```
texp MassSpring.Sens.Exp.xml
```

The sensitivity analysis uses a perturbation factor that is applied to the parameter. If this factor is too small, numerical problems are likely to occur and will lead to useless information unless specific integrators are used. So, in the case of the model in use, this perturbation factor can be set to 1e-002, where numerical problems do not appear.

```
<Prop Name="PertFactor" Value="1e-002"/>
```

A new section named `<Funcs>` has now appeared. This section calls for a quantity present in the `<Quantities>` section, so the tag `<Func Name=".x_out"/>` needs to be added here. The `texp` tool must be run one last time.

In the properties of the function, the time interval indicates at which time points the sensitivity will be calculated. So, this value must be small enough to describe well the system. A value of 0.1 is sufficient for this experiment. The other modification needed is in the `<Output Name="*File*">` section where a filename must be entered and the `Enable` option must be switched to true, as in the following tag:

```
<File Name="MassSpring.Sens.Func.out.txt" Enabled="true">
```

The experiment is now ready to be performed.

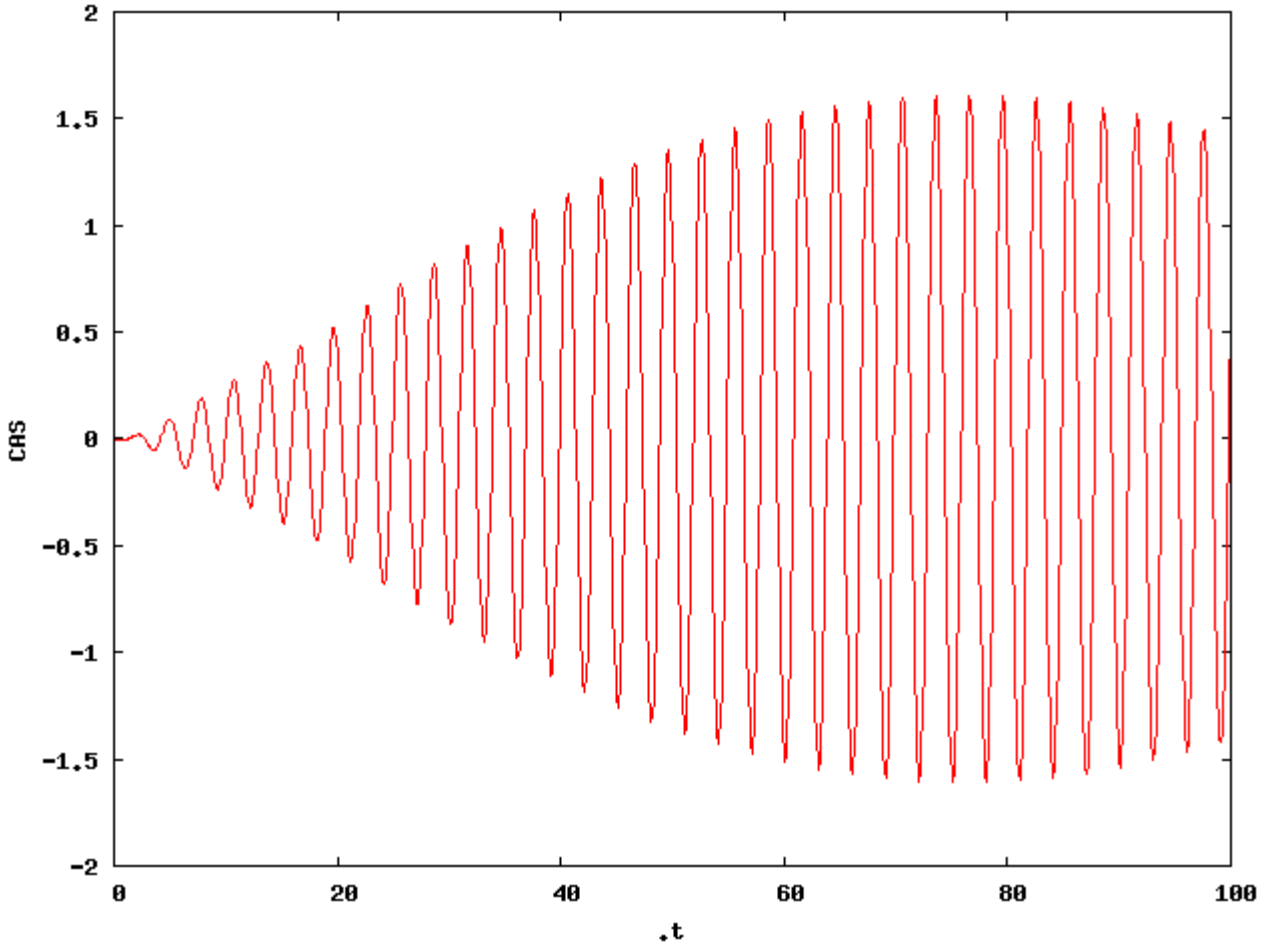
```
texec MassSpring.Sens.Exp.xml
```

The first information the user gets is that the sensitivity information is NOT reliable, followed by the expression `.P/.x_out`. But simple logic helps to remember that `x_out` crosses zero once every 1.5 units of time. So dividing any number by almost zero will lead to almost infinity.

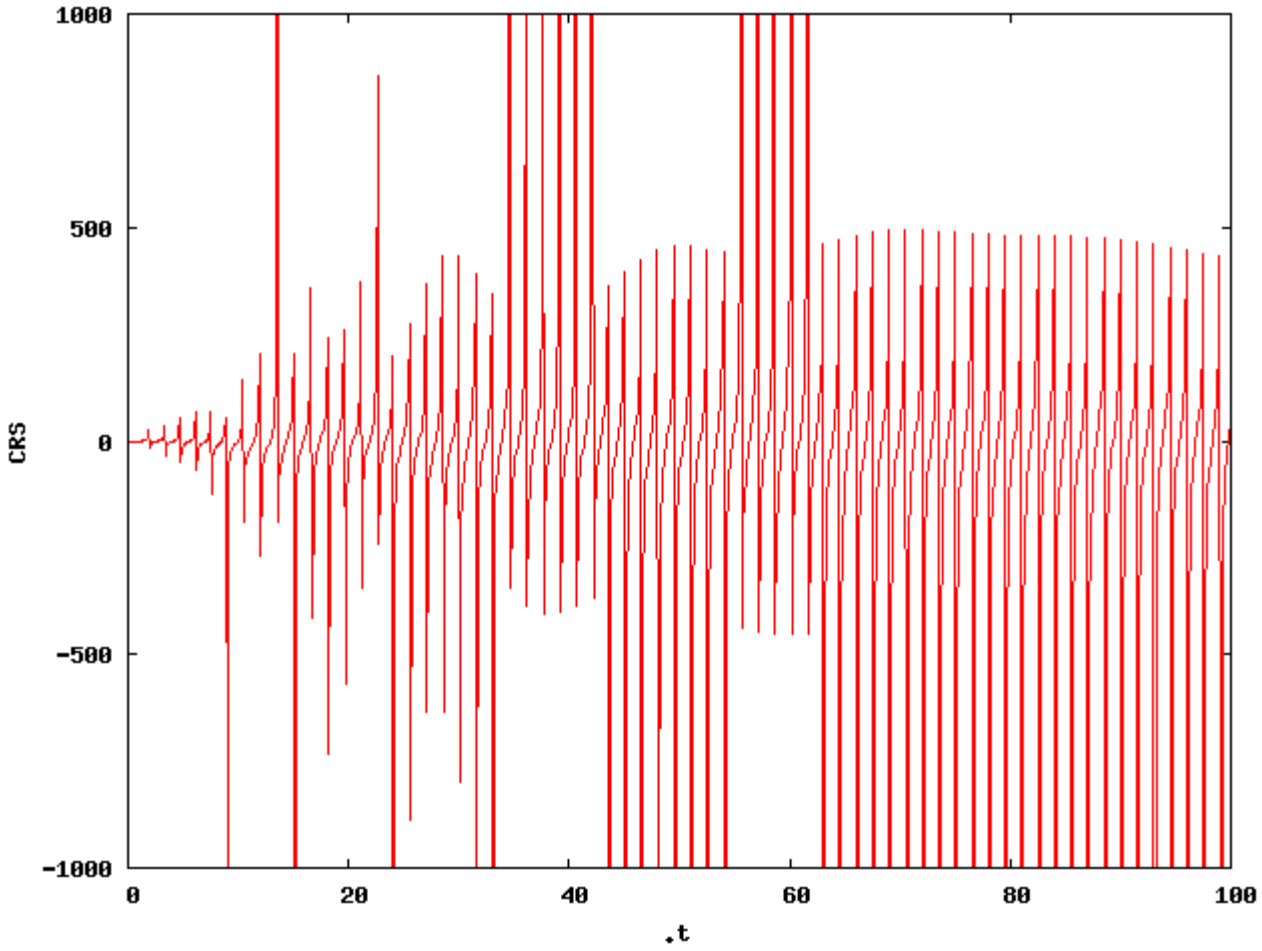
At this point, it is important to remember that Tornado is supposed to work with water quality models. So the sensitivity analysis uses criteria specifics to the water domain. In the case of a Mass-spring system, others criteria must be seek to determine if the model is reliable. However, it is not the scope of this tutorial to propose such criteria.

To gain more knowledge about sensitivity, it is useful to have a look at the `MassSpring.Sens.Func.out.txt` file. By plotting Backward, Central or Forward Absolute Sensitivity (BAS, CAS or FAS), a sinusoidal curve appears. By plotting any Relative Sensitivity (BRS, CRS or FRS) it is rather a tangent function that is revealed. And this is where the sensitivity is said to be unreliable.

The two following figures show the Central Absolute Sensitivity:



And the Central Relative Sensitivity:



Scenario Optimization

Needed Experiment: Optimization

This experiment is a combination of Optimization and Scenario experiments. Its main aim is to provide an alternative to global optimization algorithms such as `GA` and `SA`. The experiment will simply launch a succession of Optimization runs from different initials parameters nested in a Scenario experiment.

The Scenario Optimization is created by the following command:

```
tmain ExpCreateScenOptim MassSpring.Optim.Exp.xml .
```

In order to find the resonance, the tag `<Var Name=".P"/>` must be added to the `<Vars>` section toward the end of the file. A solver must also be defined for the Scenario part. For this, the following tag can be added to the `<Solve>` section:

```
<Scen Method="Plain"/>
```

In order to get the results, one needs to fill the section `<Output Name="*File*">` of the Scenario Optimization part by giving a name to the file and by switching `Enable` to `true`.

```
<File Name="MassSpring.ScenOptim.out.txt" Enabled="true">
```

Then, after running `texp` on the experiment, the following tags can be set for the variable `.P`:

```
<Prop Name="RefValue" Value="10"/>
<Prop Name="LowerBound" Value="1"/>
<Prop Name="UpperBound" Value="100"/>
<Prop Name="NoValues" Value="3"/>
```

Since this experiment should be used when minimal prior knowledge is available, the Optimization experiment should be allowed to try various values of `.P`. To do so, the values which appear in the embedded Optimization experiment should be changed to the same values as the Scenario Optimization and the constrained option of the `Simplex` algorithm has to be switched to `false`.

```
<Var Name=".P">
  <Prop Name="RefValue" Value="10"/>
  <Prop Name="LowerBound" Value="1"/>
  <Prop Name="UpperBound" Value="100"/>
<Optim Method="Simplex">
  <Prop Name="Constrained" Value="false"/>
```

In this way, three optimizations will be launched with values of `.P` between 1 and 100 in order to try to find the resonance period of the system.

Monte Carlo

Needed Experiment: Objective Evaluation

Taking his name from one of the most famous casinos of the world, this experiment relies on random numbers to set the value of parameters.

Once again, the first step is to create the experiment XML file with `tmain`:

```
tmain ExpCreateMC MassSpring.ObjEval.Exp.xml .
```

It is once again necessary to add the variables that we want to be studied in the `<Vars>` section. It is however possible to add some complexity to the function by letting also the parameter `.K` vary independently:

```
<Var Name=".K"/>
<Var Name=".P"/>
```

Then, the `texp` command will show the available tags. One can then set these values for each parameter:

```
<Var Name=".K">
  <Props>
    <Prop Name="RefValue" Value="100"/>
    <Prop Name="LowerBound" Value="90"/>
    <Prop Name="UpperBound" Value="110"/>
    <Prop Name="DistributionMethod" Value="Uniform"/>

    <Prop Name="StdDev" Value="0"/>
    <Prop Name="Values" Value=""/>
  </Props>
```

```
</Var>
<Var Name=".P">
  <Props>
    <Prop Name="RefValue" Value="3"/>

    <Prop Name="LowerBound" Value="1"/>
    <Prop Name="UpperBound" Value="5"/>
    <Prop Name="DistributionMethod" Value="Uniform"/>
    <Prop Name="StdDev" Value="0"/>
    <Prop Name="Values" Value=""/>
  </Props>
</Var>
```

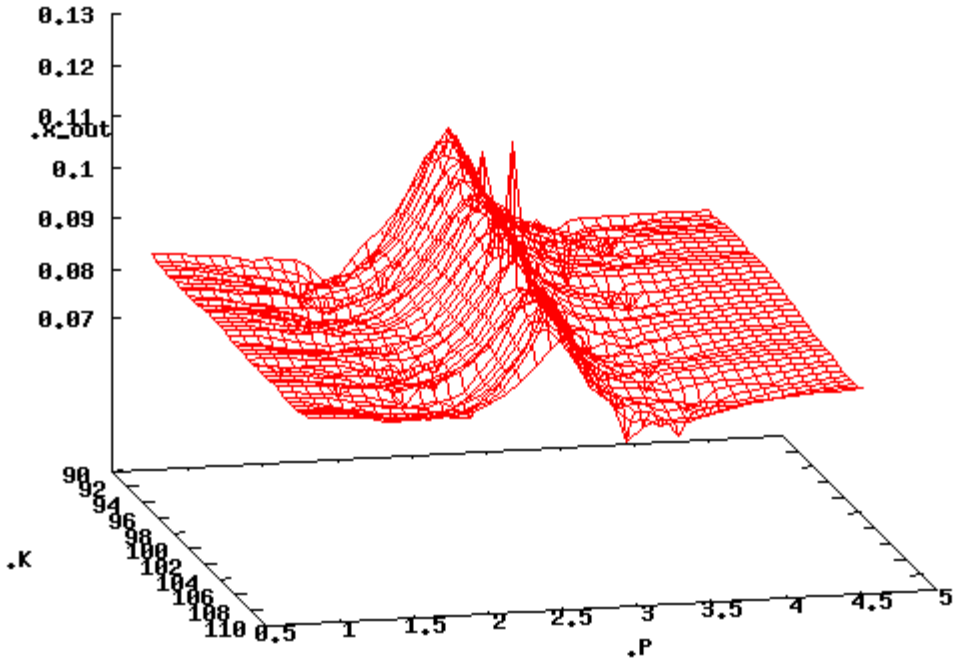
Of course, with a uniform distribution, the standard deviation has no meaning. It value must be left at zero. Finally, to have a good insight into the influence of both parameters, the number of shots must be set to a sufficiently high number. In this case, 50 shots are more than enough.

```
<MC Method="PR">
  <Props>
    <Prop Name="Generate" Value="true"/>
    <Prop Name="NoShots" Value="50"/>
```

Here is a GNUplot script that allows a 3D view of the results:

```
set dgrid3d 10,10
set xlabel ".K"
set ylabel ".P"
set zlabel ".x_out"
splot 'MassSpring.MC.out.txt' using 2:3:8 with lines
```

Although this is a random process, if the number of shots is high enough, the distribution would look like this (this figure was created with 500 shots):



Monte Carlo Optimization

Needed Experiment: Optimization

The Monte Carlo Optimization experiment combines the Monte Carlo and the Optimization experiments. Once again, the main objective of this experiment is to provide an alternative to global optimization algorithms (`GA` and `SA`) and to reduce the risk of getting trapped into local minima. So this experiment will use random values as first guess and will then try to converge to a minimum based on the specified Objective Value.

The experiment is generated as follows:

```
tmain ExpCreateMCOptim MassSpring.Optim.Exp.xml .
```

The parameter `.P` has to be added to the `<Vars>` section. Once again, there is no default algorithm given to the experiment. The solution is to manually add the following tag in the `<Solve/>` section before running `texp` on the experiment:

```
<Var Name=".P">
<MC Method="PR">
```

The number of shots `<NoShot>` is the number of tries by the experiment. In this case, ten is more than enough. Finally, the output has to be set in order to have an idea of what is being done. Here are the tags that have to be added or modified:

```
<Prop Name="NoShots" Value="10"/>
<Outputs Enabled="true">
  <Output Name="*File*">
    <File Name="MassSpring.MCOptim.out.txt" Enabled="true">
```

In order to specify the properties of `.P`, the `texp` command can be used to reveal all available options:

```
texp MassSpring.MCOptim.Exp.xml
```

To finish the preparation of the experiment, the following values have to be set under the variable `.P` in the Monte Carlo part and in the Optimization part.

```
<Prop Name="RefValue" Value="3"/>
<Prop Name="LowerBound" Value="1"/>
<Prop Name="UpperBound" Value="100"/>
<Prop Name="DistributionMethod" Value="Uniform"/>
```

The bounds are modified in order to account for the little knowledge of the model we have. Obviously, if all optimizations are started around the resonance, all guesses will succeed and the method is overkill.

In order to get better results, the step size of the Optimization algorithm should be set to 1. This value is used at the first step of the optimization algorithm. So since the optimization will occur only if the objective value varies significantly, one must ensure that this variation can be larger than the accuracy criterion. In other word, changing the step size from 0.1 to 1 increases largely the convergence region.

In this way, ten initial values of `.P` comprised between 1 and 100 will be chosen under a uniform distribution of probability in order to try to find the minimum value of the objective (here: to find the resonance of the system). The experiment is now ready, so time has come to try it out:

```
texec MassSpring.MCOptim.Exp.xml
```

A quick look at the output file `MassSpring.MCOptim.out.txt` shows that the experiment tried 10 initial guesses. Of these ten runs, five were unable to converge to the global minimum.

Statistical Analysis

Needed Experiment: Any experiment able to generate multiple data files

The Statistical Analysis can be used to compute statistics directly on multiple data files. So in order to use it, one needs to create these multiple data files. These data files are generated by the storage option available in the Objective Evaluation experiment. If all embedded experiments have been replaced by a link to the original experiment, all what is needed is to switch the storage option in the Objective Evaluation to true. If it is not the case, it is possible to manually switch to true the storage option of the Objective Evaluation embedded into another experiment.

In the case of this tutorial, the Scenario Analysis will be used to generate the multiple data files, but Monte Carlo, Optimization, Scenario Optimization or Monte Carlo Optimization would have been just fine as well.

Once the data files are generated, it might be useful to create the experiment:

```
tmain ExpCreateStats MassSpring .
```

The Statistical Analysis is very general by nature. Therefore, it does not need to be linked to any experiment. The `MassSpring` in the previous command is therefore purely arbitrary and could have been anything if one needed to specify another name.

Since the Scenario Experiment provided 15 files named `MassSpring.ObjEval.out.txt.{}` where `{}` stands for the rank of the evaluation (from 0 to 14), the statistical analysis file must be modified as follows:

```
<Prop Name="NoShots" Value="15"/>
<Prop Name="OutputFileName" Value="MassSpring.ObjEval.out.txt.{}"/>
<Prop Name="StopTime" Value="100"/>
<Prop Name="Interval" Value="0.1"/>
```

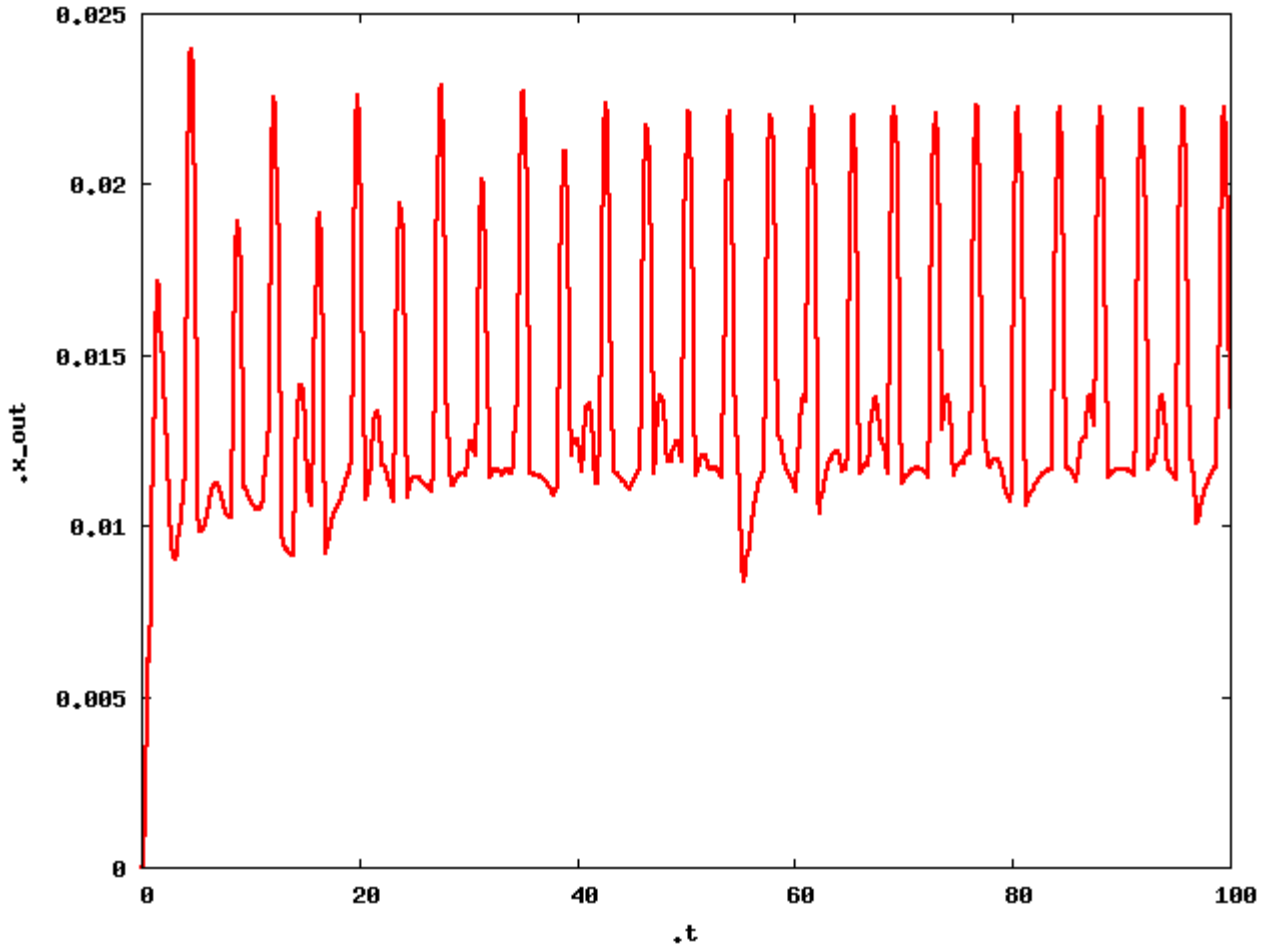
The quantity of interest `.x_out` must be added to the `<Quantities>` section with the following tag:

```
<Quantity Name=".x_out"/>
```

After expanding the file with `texp`, it is possible to select the statistics that we want to be computed:

```
<Prop Name="EnableMin" Value="true"/>
<Prop Name="EnableMax" Value="true"/>
<Prop Name="EnableAvg" Value="true"/>
```

Once the experiment is run, the results can be visualized in the `MassSpring.Stats.out.txt` file or plotted. However, in the particular case of the Mass-Spring system, only limited information can be extracted from this as is shown in the following figure. This figure shows at every time point the maximum value found in all fifteen data files of the maximum of `.x_out` with respect to time:



Ensemble Simulation

Needed Experiment: Two or more different simulations

The Ensemble Simulation Experiment is a way to perform multiple simulations and to deduce the average simulation that can be expected. In order to perform an Ensemble experiment, it is first needed to have at least two working (and different) simulation experiments. To do so, the easiest way is to copy an existing simulation and to give it a new name like `MassSpring.1.Simul.Exp.xml`. Then, to add some value to the process, this new simulation must have at least one different parameter. This parameter value can be set with the following tag in the first `<Quantities>` section:

```
<Quantity Name=".P" Value="2.5"/>
```

In this way, one doesn't need to modify the Symbolic Model to let `.P` vary. Once this is done, the Ensemble experiment can be created. Once again, this experiment is a very general experiment and does not need any pre-built experiment at the creation time:

```
tmain ExpCreateEnsemble MassSpring .
```

Once again, the `MassSpring` name is there only to give some unity to the experiments names. It could have been anything.

In the `<Exps>` section, a number of simulations can be linked. In the current case, the two following tags must be added before expanding the experiment file:

```
<Exp Name="Simul1"/>
<Exp Name="Simul2"/>
```

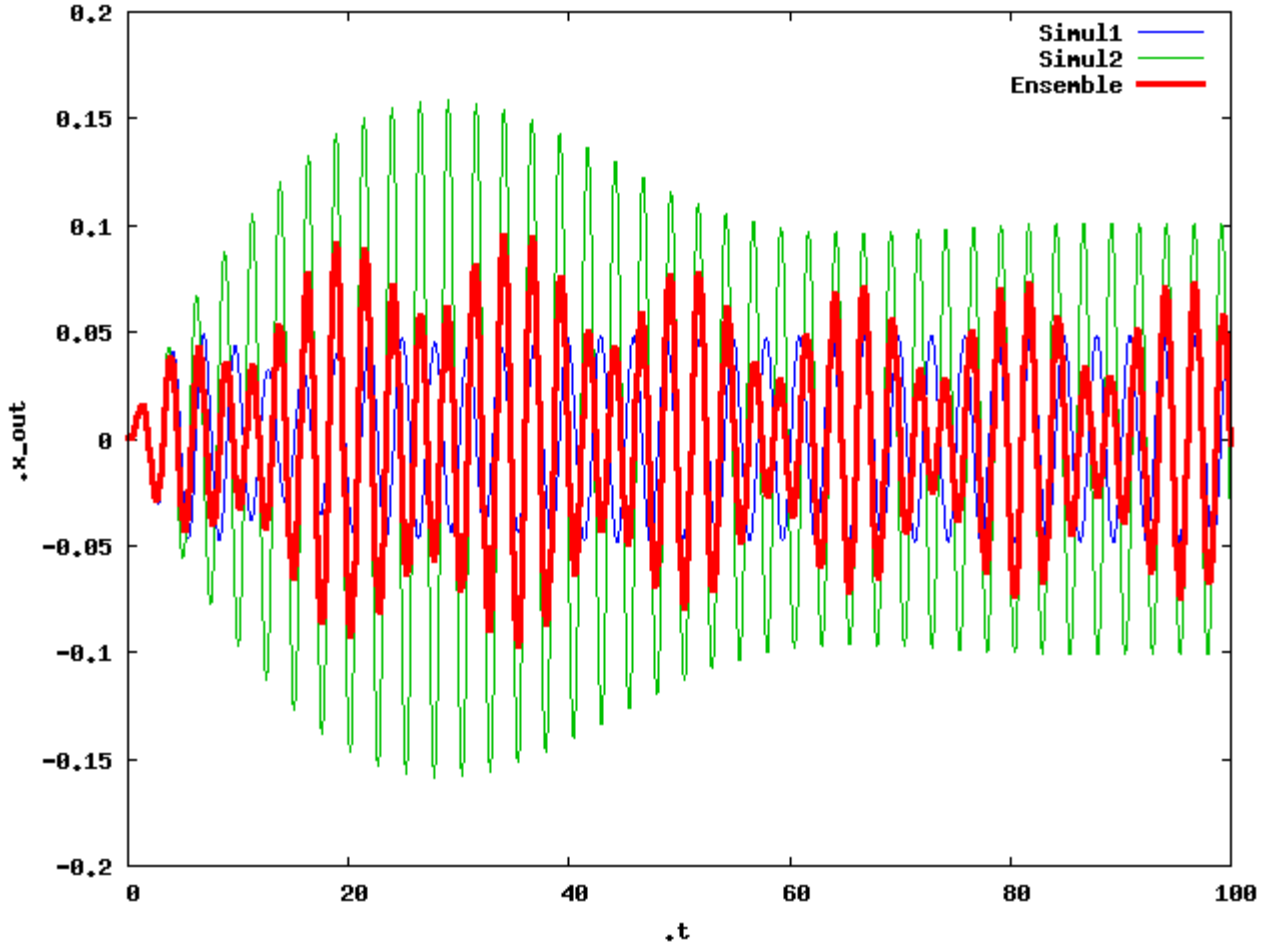
The names `Simul1` and `Simul2` are dummy names and can be replaced by anything if wanted.

Once they are expanded, the names of the simulation experiment files must be written each under its corresponding tag. The Ensemble experiment will execute the simulations and will calculate a pondered average. The weights of each simulation can be modified by adjusting the value of the `<weight>` tag embedded in the simulation to adjust. The Ensemble experiment also needs to know which quantities the user wants to see. So the tag `<Quantity Name=".x_out">` has to be added into the `<Quantities>` section. Finally, it is possible to calculate the average over only a portion of the simulation by setting `StartTime` and `StopTime` to various values and since the calculation step for each simulation may be different, the Ensemble experiment needs to have an interval on which it can calculate the average value. In the scope of the current experiment, the following values will lead to coherent results:

```
<Prop Name="StopTime" Value="100"/>
<Prop Name="Interval" Value="0.1"/>
```

In this case, since the varying parameter is the period of the extern force on the system, the resulting ensemble simulation will describe a beat between the two mass positions.

The following figure shows this beat by plotting both `Simul1` and `Simul2` with thin lines and the Ensemble with a bold line:


Simul1
Simul2
Ensemble
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
.x_out
0
20
40
60
80
100
.t

Annexe B : Représentation d'une expérience virtuelle sous Tornado

```
<?xml version="1.0" encoding="UTF-8"?>
<Tornado>
  <Exp Version="1.0" Type="ScenSSRoot">
    <Props>
      <Prop Name="TornadoVersion" Value="0.36"/>
      <Prop Name="Author" Value="PCYRIL\Cyril"/>
      <Prop Name="Date" Value="Tue Apr 21 10:54:49 2009"/>
      <Prop Name="Desc" Value=""/>
      <Prop Name="FileName" ...
            Value="PredatorPrey.ScenSSRoot.Exp.xml|.Tornado"/>
      <Prop Name="DisplayName" Value=""/>
      <Prop Name="UnitSystem" Value="Model-based"/>
    </Props>
    <ScenSSRoot>
      <Obj>
        <Exp Version="1.0" Type="SSRoot">
          <Props>
            <Prop Name="TornadoVersion" Value=""/>
            <Prop Name="Author" Value=""/>
            <Prop Name="Date" Value=""/>
            <Prop Name="Desc" Value=""/>
            <Prop Name="FileName" Value=""/>
            <Prop Name="DisplayName" Value=""/>
            <Prop Name="UnitSystem" Value="Model-based"/>
          </Props>
          <SSRoot>
            <Model Name="PredatorPrey" CheckBounds="false" ...
             StopWhenBoundsViolation="false">
              <Quantities>
              </Quantities>
            </Model>
            <Inputs Enabled="true">
            </Inputs>
            <Outputs Enabled="true">
            </Outputs>
            <Solve>
              <Root Method="Hybrid">
                <Props>
                  <Prop Name="Tolerance" Value="1e-006"/>
                  <Prop Name="MaxNoSteps" Value="200"/>
                  <Prop Name="StopCriterion" Value="Rel"/>
                </Props>
              </Root>
            </Solve>
          </SSRoot>
        </Exp>
        <Props>
          <Prop Name="TyphoonBaseName" Value="Typhoon"/>
        </Props>
```

```
      </Obj>
      <Log Name="PredatorPrey.ScenSSRoot.log.txt" Enabled="true">
        <Props>
        </Props>
      </Log>
      <Outputs Enabled="true">
        <Output Name="*Ext*">
          <Ext Enabled="false">
            <Props>
            </Props>
          </Ext>
        </Output>
        <Output Name="*File*">
          <File Name="PP.out.txt" Enabled="true">
            <Props>
              <Prop Name="DecSep" Value="."/>
              <Prop Name="Precision" Value="8"/>
            </Props>
          </File>
        </Output>
      </Outputs>
      <Vars>
        <Var Name=".pa.p">
          <Props>
            <Prop Name="RefValue" Value="10000"/>
            <Prop Name="LowerBound" Value="0"/>
            <Prop Name="UpperBound" Value="100000"/>
            <Prop Name="DistributionMethod" Value="Linear"/>
            <Prop Name="NoValues" Value="20"/>
            <Prop Name="UpperBoundPolicy" Value="IncludeUpperBound"/>
          </Props>
        </Var>
        <Var Name=".ps.p">
          <Props>
            <Prop Name="RefValue" Value="100"/>
            <Prop Name="LowerBound" Value="0"/>
            <Prop Name="UpperBound" Value="1000"/>
            <Prop Name="DistributionMethod" Value="Linear"/>
            <Prop Name="NoValues" Value="20"/>
            <Prop Name="UpperBoundPolicy" Value="IncludeUpperBound"/>
          </Props>
        </Var>
      </Vars>
      <Solve>
        <Scen Method="Seq">
          <Props>
            <Prop Name="Generate" Value="true"/>
            <Prop Name="UseTyphoon" Value="false"/>
          </Props>
        </Scen>
      </Solve>
    </ScenSSRoot>
  </Exp>
</Tornado>
```

Annexe C : Code en C++ de génération automatique des fonctions de contrainte

```
#ifdef _MSC_VER
#pragma warning(disable:4250)
#pragma warning(disable:4786)
#endif

#include "Tornado/ME/MOF2T/MOF2TAST.h"

#include "Common/Graph/Graph.h"

#include <float.h>
#include <math.h>

using namespace std;
using namespace Common;
using namespace Tornado;

void CAST::
HandleConstraintEqs()
{
  unsigned long i;
  bool Pi = false;
  wstring DummyName;
  double StartValue;

  if (!m_pRoot)
    throw CExInvMethod(__FILE__, __LINE__, L"No root");

  // Get list of DerVar's

  CST::iterator it;
  vector<CSTObject*> DerVars;
  double DefValue;

  for (it = m_pST->begin(); it != m_pST->end(); it++)
    if (it->second->GetType() == L"DerVar")
      DerVars.push_back(it->second);

  //Build constraints to each DerVar
  for (i = 0; i < DerVars.size(); i++)
  {
    DefValue = String::FromWString(DerVars[i]->GetDefaultValue(),
      L".", L",");

    //Constraints of Arctan form
    if ((DerVars[i]->GetLowerBound() != L"-INF") &&
  (DerVars[i]->GetUpperBound() != L"+INF"))
    {
      double LowerBound = String::FromWString(
```

```
  DerVars[i]->GetLowerBound(), L".", L",");

double UpperBound = String::FromWString(
  DerVars[i]->GetUpperBound(), L".", L",");

double DBound;

DBound = UpperBound - LowerBound;
StartValue = LowerBound + DBound/2;
DummyName = L"_dummy_" + DerVars[i]->GetName() + L"_";

if(Pi == false)   //To create Pi only once
{
  auto_ptr<CASTNode> pDecl(CreateDECLARATION(
    CreateID(new wstring(L"parameter")),
    CreateID(new wstring(L"Real")), CreateID(new wstring(L"Pi")),
    CreateEMPTY(), CreateEMPTY(),
    CreateNUMBER(3.141592653589793)));

  AddDecl(pDecl.release());
  m_pST->CreateAndAddCond(L"Pi", L"Param");
  Pi = true;
}

auto_ptr<CASTNode> pDecl(CreateDECLARATION(CreateEMPTY(),
  CreateID(new wstring(L"Real")),
  CreateID(new wstring(DummyName)),
  CreateATTRIBUTE(CreateID(new wstring(L"start")),
  CreateNUMBER(0.0)),
  CreateEMPTY(), CreateEMPTY()));

AddDecl(pDecl.release());
m_pST->CreateAndAddCond(DummyName, L"DerVar");

wstring DefVal = String::ToWString(0);
CST::iterator itProp;
for (itProp = m_pST->begin(); itProp != m_pST->end(); itProp++)
{
  if (itProp->first == DummyName)
  {
    itProp->second->SetDefaultValue(DefVal);
    break;
  }
}

// Dummy / (C_max - C_min)
auto_ptr<CASTNode> pArcTanArg(CreateDIV(
  CreateID(new wstring(DummyName)),CreateNUMBER(DBound)));

// atan(Dummy / (C_max - C_min))
auto_ptr<CASTNode> pArcTan(CreateFUNC(
  CreateID(new wstring(L"atan")),pArcTanArg.release()));

// (atan(Dummy / (C_max - C_min))) / (pi / 2))
auto_ptr<CASTNode> pTemp(CreateDIV(pArcTan.release(),
```

```
      CreateDIV(CreateID(new wstring(L"Pi")),CreateNUMBER(2))));

    // ((CC_max - C_min) / 2) * (atan(Dummy / (C_max - C_min))) /
       (pi / 2) + 1)
    auto_ptr<CASTNode>
      pScaleConstraint(CreateMUL(CreateNUMBER(DBound/2),
      CreatePLUS(pTemp.release(),CreateNUMBER(1))));

    // DerVar - (((CC_max - C_min) / 2) * (atan(Dummy /
       (C_max - C_min))) / (pi / 2) + 1) + C_min)
    auto_ptr<CASTNode> pFullEq(CreateMINUS(
      CreateID(new wstring(DerVars[i]->GetName())),
      CreatePLUS(pScaleConstraint.release(),
CreateNUMBER(LowerBound))));

    //Insert full equation into the AST
    auto_ptr<CASTNode> pEq(CreateEQUATION(CreateBOOLEAN(true),
      CreateID(new wstring(DummyName)), pFullEq.release()));

    pEq->SetParent(NULL);
    AddEq(pEq.release(), L"State");
    m_pST->CreateAndAddCond(L"atan", L"Function");
  }

  else
  //Building constraints of x^2 form
  {
    if (DerVars[i]->GetLowerBound() != L"-INF")
    {
      DummyName = L"_dmin_" + DerVars[i]->GetName() + L"_";
      double LowerBound = String::FromWString(
        DerVars[i]->GetLowerBound(), L".", L",");

      if(DefValue <= LowerBound)
        StartValue = 0;
      else
        StartValue = pow(DefValue - LowerBound, 0.5);

      auto_ptr<CASTNode> pDecl(CreateDECLARATION(CreateEMPTY(),
        CreateID(new wstring(L"Real")),
        CreateID(new wstring(DummyName)),
        CreateATTRIBUTE(CreateID(new wstring(L"start")),
        CreateNUMBER(StartValue)), CreateEMPTY(), CreateEMPTY()));

      AddDecl(pDecl.release());
      m_pST->CreateAndAddCond(DummyName, L"DerVar");

      DefValue = String::FromWString(
        DerVars[i]->GetDefaultValue(), L".", L",");

      if (DerVars[i]->GetLowerBound() != L"-INF")
      {
        DummyName = L"_dmin_" + DerVars[i]->GetName() + L"_";
        double LowerBound = String::FromWString(
          DerVars[i]->GetLowerBound(),L".", L",");
```

```
    auto_ptr<CASTNode> pEq(CreateEQUATION(CreateBOOLEAN(true),
      CreateID(new wstring(DummyName)),
      CreateMINUS(CreateID(new wstring(DerVars[i]->GetName())),
      CreatePLUS(CreateNUMBER(LowerBound),
      CreateDIV(CreatePOW(CreateID(new wstring(DummyName)),
      CreateNUMBER(2)),
      CreatePLUS(CreatePOW(CreateNUMBER(DefValue),
      CreateNUMBER(2)), CreateNUMBER(1)))))));

    pEq->SetParent(NULL);
    AddEq(pEq.release(), L"State");
  }
}

if (DerVars[i]->GetUpperBound() != L"+INF")
{
  double UpperBound = String::FromWString(
    DerVars[i]->GetUpperBound(), L".", L",");

  if(UpperBound <= DefValue)
    StartValue = 0;
  else
    StartValue = pow(UpperBound - DefValue, 0.5);

  DummyName = L"_dmax_" + DerVars[i]->GetName() + L"_";
  auto_ptr<CASTNode> pDecl(CreateDECLARATION(CreateEMPTY(),
    CreateID(new wstring(L"Real")),
    CreateID(new wstring(DummyName)),
    CreateATTRIBUTE(CreateID(new wstring(L"start")),
    CreateNUMBER(StartValue)), CreateEMPTY(), CreateEMPTY()));

  AddDecl(pDecl.release());
  m_pST->CreateAndAddCond(DummyName, L"DerVar");

  DefValue = String::FromWString(
    DerVars[i]->GetDefaultValue(), L".", L",");

  if (DerVars[i]->GetUpperBound() != L"+INF")
  {
    DummyName = L"_dmax_" + DerVars[i]->GetName() + L"_";
    double UpperBound = String::FromWString(
      DerVars[i]->GetUpperBound(), L".", L",");

    auto_ptr<CASTNode> pEq(CreateEQUATION(CreateBOOLEAN(true),
      CreateID(new wstring(DummyName)),
      CreateMINUS(CreateID(new wstring(DerVars[i]->GetName())),
      CreateMINUS(CreateNUMBER(UpperBound),
      CreateDIV(CreatePOW(CreateID(new wstring(DummyName)),
      CreateNUMBER(2)),
      CreatePLUS(CreatePOW(CreateNUMBER(DefValue),
      CreateNUMBER(2)),CreateNUMBER(1)))))));
    pEq->SetParent(NULL);
    AddEq(pEq.release(), L"State");
  }
```

```
      }

      wstring DefVal = String::ToWString(StartValue);
      CST::iterator itProp;
      for (itProp = m_pST->begin(); itProp != m_pST->end(); itProp++)
      {
        if (itProp->first == DummyName)
        {
          itProp->second->SetDefaultValue(DefVal);
          break;
        }
      }
    }
  }
}
```