Projet NeuralODE - Rapport intermédiaire

Apprentissage "Neuro-ODE" pour la modélisation et la commande d'un système thermique

Eva Menrad, Sébastien Gigot--Léandri, Jérôme Lalechere et Louis Bonal

31 Janvier 2024

1 Introduction

1.1 Contexte

EDF dispose d'une variété de modèles de systèmes énergétiques conçus initialement pour la simulation des procédés, souvent développés en utilisant le langage Modelica. Ce langage multiphysique, accompagné de logiciels tels que Dymola / OpenModelica, permet une modélisation détaillée des systèmes physiques. EDF a également créé des bibliothèques telles que ThermoSyspro, BuildSysPro, GridSysPro, PowerSysPro et MixSysPro pour simuler divers aspects des systèmes multi-énergies.

Ces modèles de simulation, s'avèrent complexes et ne sont pas idéaux pour le calcul de lois de commande. Notre approche dans ce projet consiste à utiliser ces modèles comme générateurs de données, en apprenant un modèle plus simple du système à partir de ces données.

L'apprentissage de modèles est une technique bien établie pour des systèmes présentant des relations statiques entre les entrées et les sorties, mais la difficulté réside dans l'apprentissage de modèles dynamiques. Nous explorerons des approches, telles que NeuroODE, pour relever ce défi.

1.2 Objectif

L'objectif de notre étude est d'étudier la possibilité d'utiliser une démarche d'apprentissage pour identifier des modèles suffisamment simples pour le calcul de lois de commande. Le système proposé comme cas d'étude est une chaudière à vapeur modélisée à l'aide de la bibliothèque ThermoSysPro.

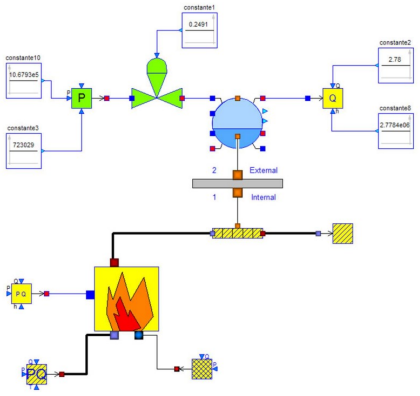

FIGURE 1 – Schéma d'une chaudière à vapeur

2 Recherche bibliographique sur l'apprentissage de systèmes dynamiques

2.1 Tour d'horizon des méthodes d'intégration, avantages et inconvénients

2.1.1 Méthode d'Euler

La méthode d'Euler est l'une des méthodes de résolution numérique d'équations différentielles les plus simples et largement utilisées. Son principe de base consiste à discrétiser un problème d'équation différentielle ordinaire (EDO) en subdivisant la trajectoire de la solution en petits intervalles de temps. À chaque pas, la méthode d'Euler estime la pente de la courbe à partir de l'état actuel du système, puis utilise cette pente pour déterminer le nouvel état du système après un petit intervalle de temps. En d'autres termes, elle approxime la courbe en utilisant une tangente locale à chaque point de la trajectoire, puis avance le long de cette tangente pour obtenir une nouvelle estimation de l'état du système. Cette procédure est répétée pour de nombreux pas de temps jusqu'à ce que la trajectoire souhaitée soit atteinte.

Etapes de la méthode d'Euler :

— Choix de la discrétisation temporelle : Divisez l'intervalle de temps t en N sous-intervalles de taille égale, où N est le nombre total de points temporels dans la discrétisation.
— Définition du pas de temps : Calculez le pas de temps h comme la largeur de chaque sous-intervalle, c'est-à-dire $h = \frac{t_{fin} - t_{debut}}{N}$.
— Initialisation : Définissez les conditions initiales de la solution. À l'instant initial t_0 , la solution y_0 est connue.
— Itérations : À chaque itération n, où n varie de 0 à $N - 1$, effectuez les étapes suivantes :

1. Évaluation de la pente : Calculez la pente $f(t_n, y_n)$ en évaluant la fonction $f(t, y)$ à l'instant actuel t_n et à la position actuelle y_n.
2. Calcul de l'incrémentation : Multipliez la pente par le pas de temps pour obtenir une estimation de l'incrément de la solution : $y_{n+1} - y_n = hf(t_n, y_n)$.
3. Mise à jour de la solution : Ajoutez cet incrément à la valeur précédente pour obtenir la nouvelle valeur de la solution à l'instant suivant : $y_{n+1} = y_n + hf(t_n, y_n)$.
4. Mise à jour du temps : Passez à l'instant suivant en augmentant le temps actuel de h, c'est-à-dire $t_{n+1} = t_n + h$.
5. Répétition : Répétez les étapes 4 pour toutes les itérations jusqu'à ce que vous atteigniez le dernier point temporel t_n.

Avantages :

— Simplicité : La méthode d'Euler est facile à comprendre et à mettre en œuvre, ce qui en fait un outil pédagogique précieux pour introduire la résolution numérique des EDO.
— Compréhension intuitive : Le concept de la méthode est intuitif, car elle suit la notion de tangente locale, ce qui facilite la visualisation des trajectoires.
— Adaptabilité : Elle peut être utilisée pour des EDO de premier ordre et est également applicable aux systèmes de plus haut ordre en les réduisant à des EDO de premier ordre.

Inconvénients :

— Précision limitée : La méthode d'Euler peut produire des approximations grossières, en particulier pour des EDO complexes ou des pas de temps relativement importants. Elle peut ne pas capturer correctement les détails de la solution.

— Sensibilité au pas de temps : La précision de la méthode dépend fortement de la taille du pas de temps. Des pas de temps trop grands peuvent entraîner des approximations inexactes et des pas trop petits des erreurs numériques.
— Instabilité pour certaines EDO : Pour certaines EDO, la méthode d'Euler peut devenir instable, conduisant à des solutions divergentes ou erronées.
— Non adaptée aux systèmes raides : La méthode d'Euler est inefficace pour les EDO raides, c'est-à-dire celles où les échelles de temps des composantes du système varient considérablement.

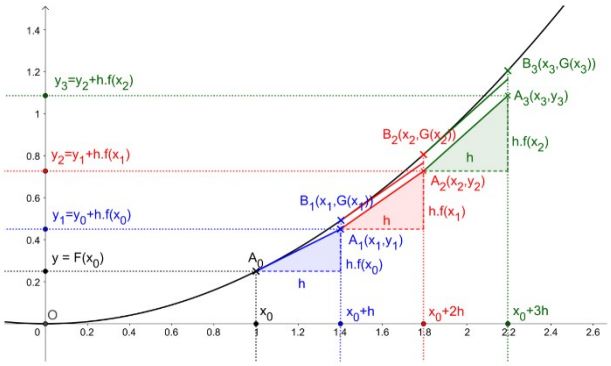

FIGURE 2 – Illustration de la méthode d'Euler explicite pour $f(x) = \frac{x^2}{2}$

2.1.2 Méthode de Runge-Kutta

La méthode de Runge-Kutta consiste à évaluer la pente de la courbe à plusieurs points intermédiaires entre le début et la fin de l'intervalle de temps. Ces pentes intermédiaires sont pondérées et utilisées pour estimer la variation de l'état du système sur l'intervalle de temps. Les méthodes de Runge-Kutta varient en fonction du nombre d'évaluations intermédiaires et des poids associés à ces évaluations. La méthode de Runge-Kutta la plus couramment utilisée est la méthode RK4, qui effectue quatre évaluations intermédiaires.

Etapes de la méthode de Runge-Kutta

— Choix de la discrétisation temporelle : Divisez l'intervalle de temps t en N sous-intervalles de taille égale, où N est le nombre total de points temporels dans la discrétisation.
— Définition du pas de temps : Calculez le pas de temps h comme la largeur de chaque sous-intervalle, c'est-à-dire $h = \frac{t_{fin} - t_{debut}}{N}$.
— Initialisation : Définissez les conditions initiales de la solution. À l'instant initial t_0, la solution y_0 est connue.
— Itérations : À chaque itération n, où n varie de 0 à $N-1$, effectuez les étapes suivantes :

1. Evaluation des pentes intermédiaires :
   — $k_1 = hf(t_n, y_n)$
   — $k_2 = hf(t_n + \frac{h}{2}, y_n + \frac{k_1}{2})$
   — $k_3 = hf(t_n + \frac{h}{2}, y_n + \frac{k_2}{2})$
   — $k_4 = hf(t_n + h, y_n + k_3)$
2. Calcul de l'incrément : Calculez l'incrément de la solution comme une moyenne pondérée des pentes intermédiaires : $y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
3. Mise à jour du temps : Passez à l'instant suivant en augmentant le temps actuel de h, c'est-à-dire $t_{n+1} = t_n + h$.

— Répétition : Répétez les étapes 4 pour toutes les itérations jusqu'à ce que vous atteigniez le dernier point temporel t_N.

Avantages :

— Précision accrue : Les méthodes de Runge-Kutta offrent une meilleure précision par rapport à la méthode d'Euler, ce qui les rend adaptées à la résolution de problèmes où la précision est essentielle.
— Robustesse : Elles sont plus robustes et stables que la méthode d'Euler, ce qui les rend adaptées à un large éventail de problèmes, y compris des EDO raides.
— Contrôle de la précision : La méthode de Runge-Kutta permet un contrôle plus précis de la précision en ajustant le nombre d'évaluations intermédiaires ou en réduisant la taille du pas de temps.

Inconvénients :

— Complexité accrue : La méthode de Runge-Kutta est plus complexe à mettre en œuvre que la méthode d'Euler en raison du calcul des évaluations intermédiaires.
— Consommation de ressources : Elle nécessite plus d'évaluations de la fonction d'état par rapport à la méthode d'Euler, ce qui peut être coûteux en termes de ressources de calcul.
— Pas de temps variable : Bien qu'elle permette un contrôle de la précision, elle peut nécessiter un ajustement du pas de temps, ce qui peut être compliqué dans certains cas.

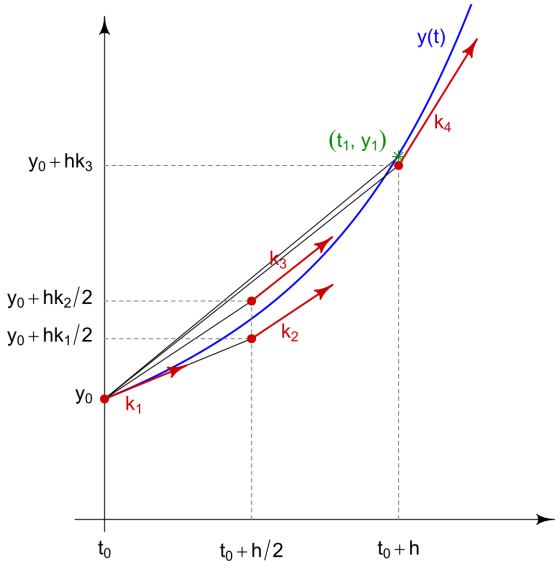

FIGURE 3 – Illustration de la méthode de Runge-Kutta d'ordre 4

2.1.3 Méthode de Heun

La méthode de Heun est une méthode numérique utilisée pour résoudre des équations différentielles ordinaires (EDO). Elle appartient à la famille des méthodes de Runge-Kutta. La méthode de Heun est d'ordre deux, ce qui signifie qu'elle a une précision d'approximation du second ordre. Elle est également parfois appelée la méthode de Runge-Kutta d'ordre deux. Cette méthode combine une estimation de pente initiale avec une correction basée sur une pente moyenne sur l'intervalle de temps considéré. Elle offre une meilleure

précision que la méthode d'Euler explicite tout en restant relativement simple à implémenter.

Etapes de l'implémentation de la méthode de Heun

— Choix de la discrétisation temporelle : Divisez l'intervalle de temps t en N sous-intervalles de taille égale, où N est le nombre total de points temporels dans la discrétisation.
— Définition du pas de temps : Calculez le pas de temps h comme la largeur de chaque sous-intervalle, c'est-à-dire $h = \frac{t_{fin} - t_{debut}}{N}$.
— Initialisation : Définissez les conditions initiales de la solution. À l'instant initial t_0 , la solution y_0 est connue.
— Itérations : À chaque itération n, où n varie de 0 à $N - 1$, effectuez les étapes suivantes :

1. Évaluation de la pente initiale : Calculez la pente initiale k_1 à l'instant t_n $k_1 = hf(t_n, y_n)$
2. Estimation provisoire : Utilisez k_1 pour effectuer une estimation provisoire y_{pred} de la solution à l'instant suivant t_{n+1} à l'aide de la méthode d'Euler explicite : $y_{pred} = y_n + k_1$
3. Évaluation de la pente corrigée : Calculez la pente corrigée k_2 à l'instant t_{n+1} en utilisant y_{pred} : $k_2 = hf(t_{n+1}, y_{pred})$
4. Calcul de l'incrément : Calculez l'incrément de la solution comme une moyenne pondérée des pentes initiale et corrigée : $y_{n+1} = y_n + \frac{1}{2}(k_1 + k_2)$
5. Mise à jour du temps : Passez à l'instant suivant en augmentant le temps actuel de h, c'est-à-dire $t_{n+1} = t_n + h$.

— Répétition : Répétez les étapes 4 pour toutes les itérations jusqu'à ce que vous atteigniez le dernier point temporel t_N.

Avantages :

— Facilité d'implémentation : La méthode de Heun est relativement simple à implémenter par rapport à des méthodes plus complexes.
— Deuxième ordre : La méthode de Heun est d'ordre deux, ce qui signifie qu'elle offre une meilleure précision que la méthode d'Euler explicite (d'ordre un).
— Stabilité : La méthode de Heun est généralement stable pour une gamme raisonnable de pas de temps, ce qui en fait un choix robuste dans de nombreux cas.

Inconvénients :

— Précision limitée : Bien que d'ordre deux, la méthode de Heun peut encore manquer de précision dans certaines situations par rapport à des méthodes d'ordre supérieur, notamment pour des problèmes raides.
— Pas de temps fixe : Comme beaucoup de méthodes numériques, la méthode de Heun utilise un pas de temps fixe, ce qui peut être inefficace pour les problèmes où les échelles de temps varient considérablement.
— Sensibilité aux variations de la fonction : La méthode de Heun peut être sensible aux variations rapides de la fonction à résoudre, ce qui peut entraîner une précision réduite dans ces cas.

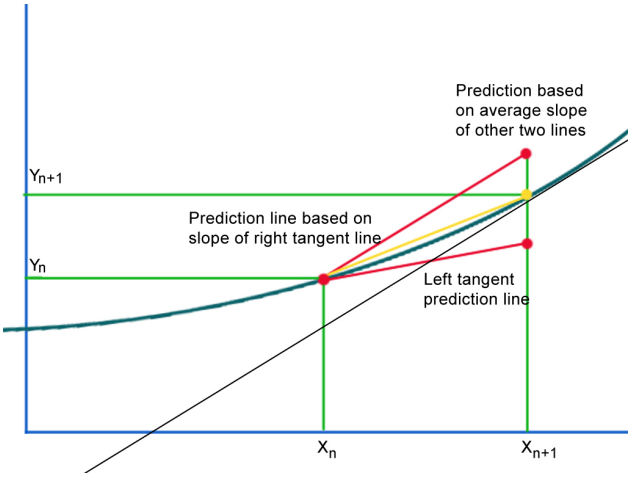

FIGURE 4 – Illustration de la méthode de Heun

2.2 Les réseaux Neural-ODE

2.2.1 Principes des NODEs

Les Neural-ODE, ou NODEs, sont une classe de réseaux de neurones récurrents faisant partie des architectures de réseaux de neurones continus : ils se basent sur la prédiction de la dérivée de l'état plutôt que de la prédiction à temps fixe et discret, ce qui, en utilisant une méthode d'intégration d'EDO tel que RK4, permet donc d'obtenir un modèle avec une grande adaptibilité, puisque la complexité du programme devient relative au temps de discrétisation de la méthode d'intégration et non au nombre de couches du RNN. Contrairement au modèle plus classique comme les ResNet, la précision de la prédiction et le temps de calcul est donc relié au vecteur des temps discrets donné en entrée pour la méthode d'intégration, et peut donc varier entre chaque problème, voir entre chaque itération, sans pour autant nécessité aucun apprentissage supplémentaire (5).

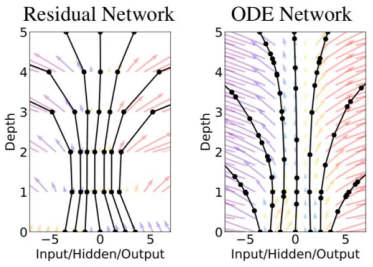

FIGURE 5 – Différence entre ResNet et NODE [2]

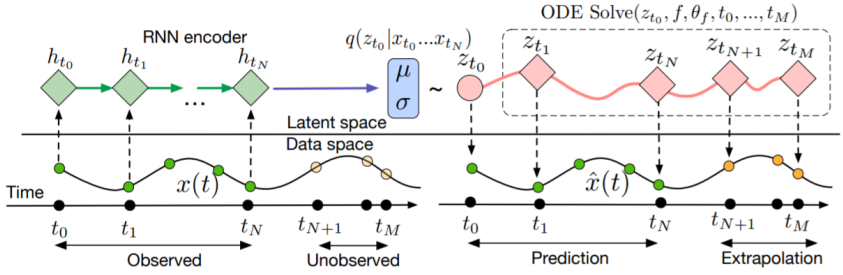

FIGURE 6 – Principe du modèle de prédiction NODE [2]

La principale difficulté de ce modèle de couches continues et la rétropropagation ou backpropagation, mécanisme d'apprentissage des réseaux de neurones dans laquelle le gradient de la fonction de coût est évalué pour chaque couche afin d'améliorer celle-ci à la prochaine itération. Dans un RNN fini, cette étape est simple et peu côuteuse en restant dans des proportions de nombres de paramètres raisonnables, mais pour les réseaux de neurones continues, en raison des approximations numériques et du coût en mémoire important de la différentiation de la méthode d'intégration, l'utilisation direct de l'algorithme de rétropropagation adapté aux NODEs peut se révéler problématique. Les NODEs utilisent donc souvent la méthode de l'adjoint [6]

pour optimiser l'erreur de prédiction. Dans le cadre de la librairie utilisée (torchdiffeq) dans notre projet, la méthode de rétropropagation classique comme la méthode de l'adjoint ont été implémentées et la différence d'efficacité entre ces deux approches pourra donc être évaluée.

Les NODEs sont donc une architecture avec un important potentiel pour la prédiction et la génération de série temporelle. Malgré des difficultés de convergence observés, plusieurs utilisations concrètes ont été envisagées. Le projet APHYNITY [1], sur lequel se base également se projet, étudie la convergence d'un modèle hybride sommant l'apport d'un module analytique et d'un module basé sur une approche NODEs pour modéliser l'évolution de systèmes simples. Des approches permettant le contrôle optimal, via algorithme d'entropie croisée pour traiter des non-linéarités du modèle de MPC utilisé, ont également était étudiées pour exploiter le potentiel des NODEs dans le domaine de l'optimisation de la commande [4].

3 Plan et objectifs du projet

Après notre étude bibliographique sur le concept de NeuralODE, de FMU et des différentes méthodes d'intégration, nous avons au fur et à mesure extrait 2 voies de progression parallèles pour le projet :

- Se familiariser avec OpenModelica
- Programmation de cas jouets
- Implémenter un RNN sous OpenModelica
- Créer un algorithme de commande

- Se familiariser avec les NODEs via le projet Aphynity
- Coder et tester des exemples simples pour l'apprentissage
- Implémenter des cas plus complexes avec apprentissage d'entrées

Un premier aspect du projet et en effet l'utilisation d'un langage de programmation permettant de simuler différents systèmes. Ce langage est Modelica et nous utlisons OpenModelica pour simuler nos cas simples (et plus tard faire tourner un modèle de chaudière). Lier OpenModelica à un réseau de neurone type NeuralODE afin de contrôler la chaudière sera un des défis du projet. La stratégie de commande sera basé sur de la commande prédictive comme cela a déjà été étudié.[4]

Du côté du développement et de l'entrainement d'un réseau de neurones récurrent, nous avons opté pour une utilisation de Python et de la librarie Pytorch. Le projet APHYNITY [1], visant à développer une approche hybride ML/EDO pour la prédiction de série temporelle, a été un pilier important pour l'implémentation de notre modèle. Le code qui en résulte et donc un programme se basant sur la génération de données à partir d'un modèle défini, et entrainant un RNN sur la prédiction de ses données en se basant sur les conditions initiales de l'échantillon.

Le diagramme ci-dessous affiche la progression de notre projet concercant les deux voies principales mentionné, celle suivant le développement de l'architecture de contrôle sous Modelica (violet) et l'autre concernant la création et l'entrainement du NODEs.

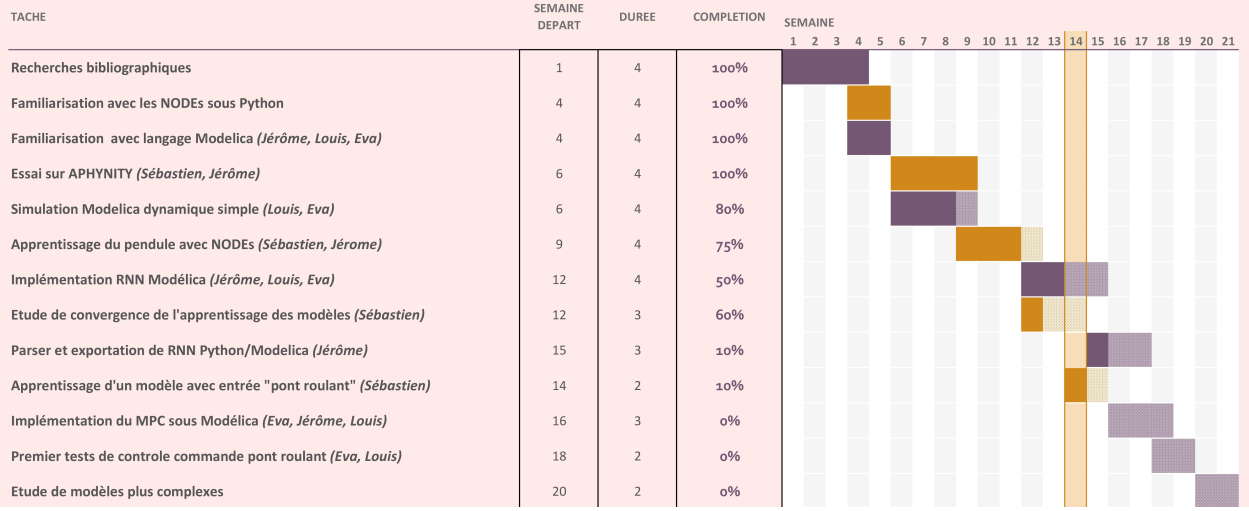

FIGURE 7 – Diagrammme de planification du projet suivant les différents degrés d'avancement des tâches et de leur position temporelle dans notre étude.

4 Tests sur des cas simples

4.1 Pendule simple

4.1.1 OpenModelica

Un des systèmes mécaniques les plus simples est le pendule simple. Pour les premiers tentatives, ce simple modèle est utilisé. La langage Modelica offre des exemples de base. Le pendule simple est un de ces exemples et est composé par un champ de gravitation, un armortisseur, une articulation et une masse comme il montre la figure 10. Pour les pendule simple sans frottement le coefficiet d'amortissment d de l'amortisseur est choisi à zéro. La simulation du pendule est affichée en figure 9. L'amplitude de l'accélération du pendule ne se réduit pas à cause de l'absence du frottement.

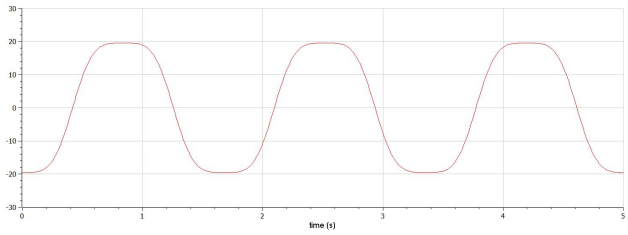

FIGURE 8 – Accélération angulaire du pendule simple sans frottement sur OpenModelica

4.1.2 Neural-ODE

La convergence pour le pendule simple sans frottement n'est pas bonne. Elle n'est pas possible en data-driven et est mauvaise en utilisant un mode hybride comme dans l'expérience Aphynity.

4.2 Pendule simple avec frottement

Un cas simple étudié par une des expériences d'Aphynity est le pendule simple avec frottement.

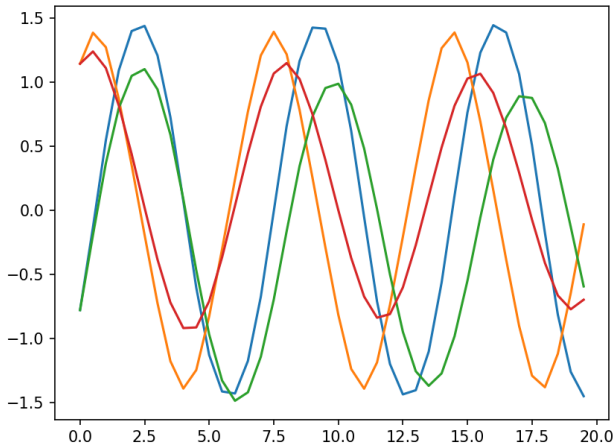

FIGURE 9 – Solution obtenue avec un algorithme hybride (une partie modèle et une partie data-driven)

4.2.1 OpenModelica

Modelica fournit un pendule simple avec frottement dans sa librairie de base.

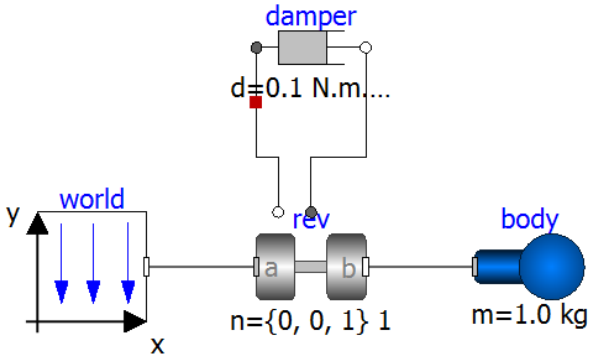

FIGURE 10 – Schéma du pendule simple avec frottement sur Openmodelica

Une simulation de ce modèle donne la courbe suivante.

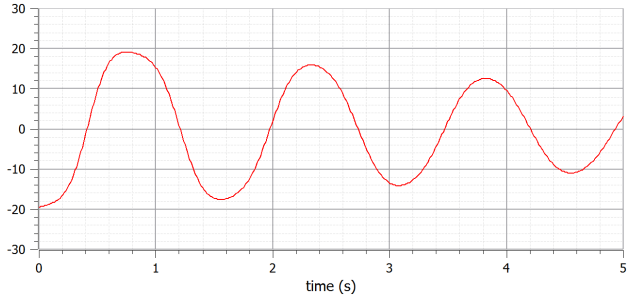

FIGURE 11 – Simulation du pendule simple avec frottement sur Openmodelica

4.2.2 Neural-ODE

Nous sommes capables d'avoir une bonne convergence avec un modèle entièrement data-driven.

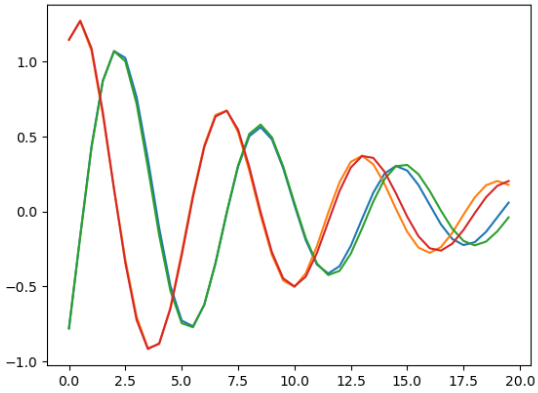

FIGURE 12

Cette convergence est assez robuste puis qu'elle fonctionne pour différentes valeurs d'amortissement et différents pas temporel.

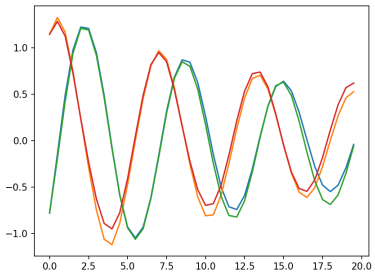

FIGURE 13 – $\alpha = 0,1$

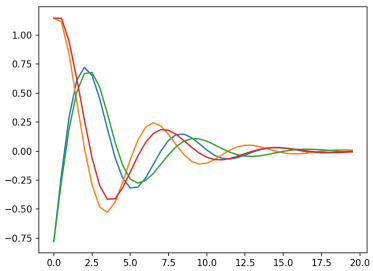

FIGURE 14 – $\alpha = 0,5$

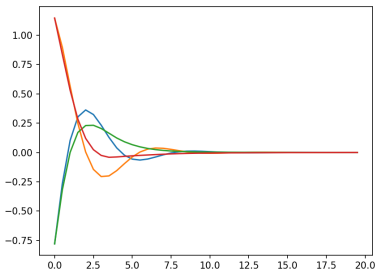

FIGURE 15 – $\alpha = 1$

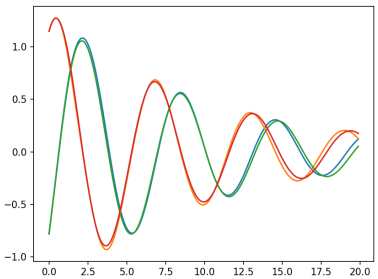

FIGURE 16 – $dt = 0,1s$

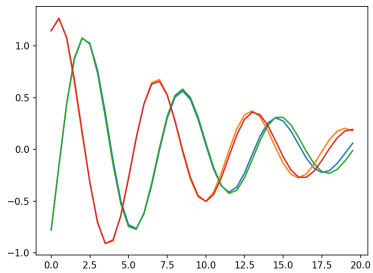

FIGURE 17 – $dt = 0,5s$

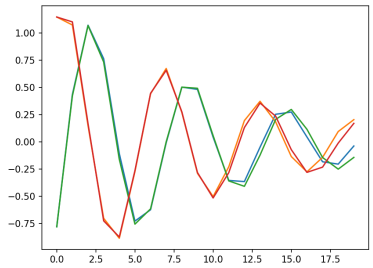

FIGURE 18 – $dt = 1s$

5 Perspectives

5.1 Connexion Openmodelica & Python

La transmission du réseau de neurone généré en Python dans le modèle sous Openmodelica pour réaliser nos simulations est primordial au sein de notre projet.

Pour réaliser ce lien, plusieurs méthodes s'offrent à nous, notamment les suivantes :

— **L'utilisation de FMU (Functional Mock-up Units) :** Les FMU, acronyme de "Functional Mock-up Units" ou "Unité de Modélisation Fonctionnelle" en français. Ces fichiers contiennent des modèles de simulation, souvent sous la forme de composants logiciels, facilitant ainsi l'intégration de divers modèles au sein d'environnements de simulation étendus.

Leurs principals atouts sont l'indépendance et la réutilisabilité. En effet ils sont conçus pour l'être. Cette caratéristique permet de les créer et de les tester de manière autonome avant leur intégration dans des environnements de simulation plus vastes.

Structure d'une FMU :

— **Code exécutable :** C'est le code responsable de la logique de simulation. Ce code peut être fourni sous forme de bibliothèque dynamique ou de manière autonome, dans n'importe quel langage compilable en code machine.

— **Fichier XML :** C'est le fichier décrivant le modèle de simulation, notamment sa structure et ses caractéristiques. Il peut être considéré comme le coeur du FMU, en effet il décrit les détails sur les entrées, les sorties, les paramètres, les unités de l'ensemble du modèle de simulation.

— **Fichiers supplémentaires :** Comprend les éléments nécessaires à l'exécution du code, comme des bibliothèques de fonctions ou des fichiers de configuration.

— **Ressources nécessaires :** Ce sont les données d'initialisation ou les données externes essentiels à l'exécution du modèle.

En revanche, l'utilisation des FMU présentent tout de même des inconvénients, en effet la complexité de leur création, les difficultés potentielles d'intégration aux logiciels que nous allons utiliser et le surcoût en terme de taille de fichier, doivent être pris en compte lors du choix de notre méthode.

— **Utilisation de code externe :** Consiste en la conversion du modèle de simulation en code compilable dans notre cas en Python. L'approche est similaire à celle vu avec les FMU, en revanche elle présente l'avantage d'être moins strict, mais elle nécessite une bonne connaissance du système étudié afin de réaliser la conversion du code d'un langage à un autre sans dénaturer le système étudié.

— **Implémentation du réseau de neurone dans Openmodelica :** Avec cette méthode, la structure du réseau de neurone peut-être directement sauvegardé dans Openmodelica. Chaque couche de celui-ci peut-être représenté par un modèle sous forme de code ou de bloc (dans la représentation visuelle) avec des fonctions d'activations prédéfinis.

La définition du poids et des biais de chaque couche du réseau peut-être conservés au sein de code Modelica ou bien directement au sein d'un fichier externe.

Avantage principal de cette méthode est l'absence de problème de mapscompatibilité et permet une représentation graphique du réseau de neurone utilisé qui peut aider à la compréhension et l'interprétation des résultats. En revanche, de part ces caractéristiques, cette méthode s'adapte moins facilement à des évolutions régulières de la structure du réseau de neurones et réduit l'adaptabilité.

6 Références

[1] Yuan Yin et al. « Augmenting Physical Models with Deep Networks for Complex Dynamics Forecasting ». In : *Journal of Statistical Mechanics : Theory and Experiment* 2021.12 (déc. 2021), p. 124012. ISSN : 1742-5468. arXiv : 2010.04456 [cs, stat].

[2] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. Déc. 2019. arXiv : 1806.07366 [cs, stat].

[3] Zakariae EL Asri. *Physics and Model-Based Reinforcement Learning*.

[4] Zakariae EL Asri et al. « Residual Model-Based Reinforcement Learning for Physical Dynamics ». In : *3rd Offline RL Workshop : Offline RL as a "Launchpad"*. Oct. 2022.

[5] Stefano Massaroli et al. *Dissecting Neural ODEs*. Jan. 2021. arXiv : 2002.08071 [cs, stat].

[6] L. S. Pontryagin. *Mathematical Theory of Optimal Processes*. CRC Press, mars 1987. isbn : 978-2-88124-077-5.

[7] F Codeca et F Casella. *Neural Network Library in Modelica*. 2006.