Simulation d'un convertisseur d'électronique de puissance avec le language Julia

Florian DUPRIEZ-ROBIN
CEA-Tech Pays de la Loire, Technocampus Océan, 5 Rue de l'Halbrane, Bouguenais, 44340, France

RESUME – La simulation de l'électronique de puissance par un système de calcul numérique permet de mieux comprendre les convertisseurs électriques. Cette simulation est actuellement réalisée par différent langage de programmation. Ce travail permet d'ajouter un nouveau langage de programmation pour cela : Julia[1]. Julia est un langage de haut niveau, open-source, performant, dynamique, pour le calcul scientifique. Il permet une utilisation simple de l'exécution parallèle ou distribuée.

Mots-clés—Électronique de puissance, Julia, simulation, modélisation.

1. INTRODUCTION

La simulation d'électronique de puissance est nécessaire en phase de développement, que ce soit pour l'évaluation d'un montage en électronique de puissance [2] ou pour celle d'un contrôle commande de ce système. Cette simulation permet de valider un montage avant de réaliser et de tester ce montage réellement.

Pour modéliser un tel convertisseur, deux grandes familles de résolutions sont possibles : les modélisations causales et les modélisations acausales.

La modélisation causale permet de définir la relation entre les entrées d'un système (émetteur d'énergie) et les sorties de ce système (consommateur d'énergie) en présupposant que cette relation n'est pas réversible. Ce type de modélisation permet une résolution rapide et efficace du fonctionnement d'un système mais nécessite un effort de conception important et n'est représentatif que d'un type de réalité : quand le flux d'énergie transite des entrées vers les sorties du système.

La modélisation acausale permet de définir la relation entre les différents éléments d'un système sans présupposer le sens des transferts énergétiques au sein de ce système. Ce type de modélisation est plus simple à représenter mais le système de résolution est plus complexe.

Dans cet article nous étudierons une méthode de résolution acausale.

Différents langages de programmation permettent une résolution acausale de système électrique comme Matlab/Simulink avec la boite à outils Simscape, Modélica et d'autres. Ces boites à outils sont complètes, simple d'utilisation avec une programmation passant par une interface graphique. Ces outils ont pour défaut d'être payant pour le premier, et ne peuvent pas utiliser simplement les outils informatiques moderne : le calcul en parallèle sur plusieurs cœurs ou plusieurs serveurs et le calcul sur carte graphique (GPU).

C'est pour résoudre cette difficulté que ces travaux préliminaires de modélisation et de simulation d'un convertisseur d'électronique de puissance par le language Julia ont été effectués.

Julia est un langage de programmation haut niveau, créé par J. Bezanson, S. Karpinski, V. B. Shah, et A. Edelman [1]. L'objectif de ce langage est de permettre le calcul scientifique haute performance. L'objectif lors de sa création est de se coder comme un programme Python, et de s'exécuter aussi vite qu'un programme compilé en C. Les lignes de code de cet article vous prouveront sa simplicité de programmation, des nombre articles présentant des tests massifs prouvent que le second objectif aussi atteint [3].

Le convertisseur est modélisé puis simulé sous Julia est un modèle simple de convertisseur abaisseur de tension.

La référence à l'état de l'art utilisés pour cette simulation sera Matlab/Simulink, avec la boite à outils Simscape Electrical.

2. MODELISATION D'UN CONVERTISSEUR D'ELECTRONIQUE DE PUISSANCE

Un convertisseur d'énergie électrique peut être vu comme un système multipôles ayant en entrée une ou plusieurs sources d'énergies électriques et en sortie un ou plusieurs consommateurs.

Le convertisseur étudié durant cette étude est présenté en figure 1.

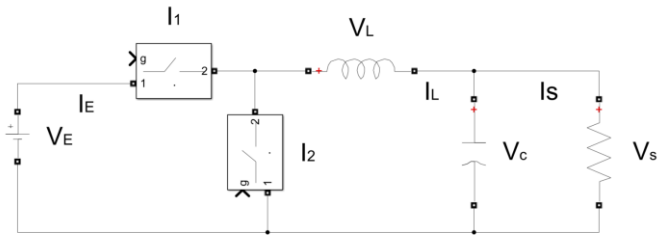

Fig. 1. Modèle du convertisseur abaisseur étudié.

Les composants du circuit sont :

- Une source de tension idéale notée V_E;
- Deux interrupteurs idéaux notés I_1 et I_2 ;
- Une inductance idéale notée V_L ;
- Un condensateur idéal nommé V_C ;
- Une résistance idéale nommée V_S ;

2.1. *Définition des variables d'états*

Pour modéliser ce convertisseur, il faut distinguer deux types de composants :

- Les composants dont l'état actuel ne dépend pas de l'état précédent, comme une résistance dont l'équation est $V_R = R * I_R$ avec V_R la tension au borne de la résistance, I_R son courant et R sa valeur en Ohm.
- Les composants dont l'état actuel dépend de l'état précédent, comme un condensateur dont l'équation à résoudre est : $I_c = C * \frac{dV_C}{dt}$ V_c et I_c la tension et le courant du condensateur V_C de valeur C.

Un circuit électrique n'ayant que des composants ne dépendant pas du temps peut être simulé très simplement : le circuit pourra être résumé à une équation directe. Un exemple : le pont diviseur de tension, composé de 2 résistances en série et qui permet de diviser la tension d'entrée par le ratio des 2 résistances.

Dès qu'il y a un composant dont l'état actuel dépend de l'état précédent, il faut résoudre des équations différentielles. La variable devant être dérivée sera appelé variable d'état. Dans le cas du condensateur la variable d'état est la tension V_C. C'est pour ce type de composant qu'il faut un logiciel performant pour simuler le comportement d'un circuit électrique.

2.2. *Modélisation d'un convertisseur d'électronique de puissance*

La conversion d'énergie électrique par des interrupteurs de puissance se base sur l'énergie stockée dans les condensateurs et les inductances. Pour définir l'état du convertisseur étudié il faut résoudre les équations suivantes :

$$\begin{aligned} V_L &= L * \frac{dI_L}{dt} \\ I_c &= C * \frac{dV_C}{dt} \end{aligned} \quad (1)$$

Avec V_L et I_L la tension et le courant de l'inductance V_L de valeur L. V_c et I_c la tension et le courant du condensateur V_R de valeur C. I_L et V_c sont les variables d'état de notre système. À partir de ces modèles de composants, nous pouvons mettre en équation un convertisseur sous la forme suivante :

$$\begin{cases} \dot{X} = A * X + B * U + X_0 \\ Y = C * X + D * U \end{cases} \quad (2)$$

Avec X le vecteur des variables d'état du convertisseur étudié, $\dot{X}$ la dérivée de ce vecteur, U le vecteur d'entrée du système, X_0l'état initial du système, Y le vecteur de sortie du système, A, B,C et D les matrices de transfert du convertisseur.

2.3. *Modélisation du hacheur dévolteur*

Dans le cas présent (voir figure 1) nous étudierons le modèle le plus simple : une source d'énergie électrique et un consommateur. Ce convertisseur peut donc être résumé à un système quadripôle comprenant une entrée ayant deux pôles, V_e et une sortie ayant deux pôles V_s .

Les hypothèses de modélisations sont les suivantes :

- les interrupteurs d'électronique de puissance du modèle sont idéaux : pas de pertes à leurs bornes et une commutation instantanée ;
- la source d'énergie est idéale : pas de résistance interne et une puissance infinie ;

La fonction de transfert du quadripôle étudié dépend de l'état de ses interrupteurs. Ceux-ci peuvent être dans 2 états : soit bloqué, soit passant. Il y a 2^n matrices de transfert dans un convertisseur d'électronique de puissance. Dans notre cas $2^2 = 4$ fonctions de transfert différentes. Dans ces 4 états, l'état ou I_1et I_2 sont passants, ne sera pas étudié car cela revient à mettre l'entrée en court-circuit. Nous avons donc comme fonction de transfert possible :

$$\begin{cases} \begin{matrix} V_C \\ I_L \end{matrix}^{\cdot} = \begin{bmatrix} -\frac{1}{C*R_S} & \frac{1}{C} \\ 0 & -\frac{1}{L} \end{bmatrix} * \begin{matrix} V_C \\ I_L \end{matrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} * V_e & I_1 \ passant \\ V_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} * \begin{matrix} V_C \\ I_L \end{matrix} & I_2 \ bloqué \end{cases} \quad (3)$$

$$\begin{cases} \begin{matrix} V_C \\ I_L \end{matrix}^{\cdot} = \begin{bmatrix} -\frac{1}{C*R_S} & \frac{1}{C} \\ 0 & -\frac{1}{L} \end{bmatrix} * \begin{matrix} V_C \\ I_L \end{matrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} * V_e & I_1 \ bloqué \\ V_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} * \begin{matrix} V_C \\ I_L \end{matrix} & I_2 \ passant \end{cases} \quad (4)$$

$$\begin{cases} \begin{matrix} V_C \\ I_L \end{matrix}^{\cdot} = \begin{bmatrix} -\frac{1}{C*R_S} & 0 \\ 0 & 0 \end{bmatrix} * \begin{matrix} V_C \\ I_L \end{matrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} * V_e & I_1 \ bloqué \\ V_s = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} * \begin{matrix} V_C \\ I_L \end{matrix} & I_2 \ bloqué \end{cases} \quad (5)$$

La conversion sera continue, le système ne sera jamais dans l'état de l'équation (5). Le système dans l'état (3) sera appelé passant et dans l'état (4) bloqué.

La tension de sortie va dépendre du $T_{passant}$, durée à l'état (3) et $T_{bloqué}$ la durée à l'état (4). Le rapport cyclique $\alpha = \frac{T_{passant}}{T_{passant}+T_{bloqué}}$, et la période est $P = T_{passant} + T_{bloqué} = 1/f$ avec f la fréquence de commutation. Ce type de commutation est une Modulation de Largeur d'Impulsion (MLI), ou PWM en anglais.

Nous pouvons maintenant programmer le modèle sous Julia.

3. PROGRAMMATION SOUS JULIA

Julia est un langage de haut niveau, la programmation est simple et explicite.

Pour la simulation d'un convertisseur d'électronique de puissance, il faut résoudre deux problèmes :

- Modifier la fonction de transfert du système en fonction de l'état des interrupteurs de puissance.
- Résoudre les équations différentielles de la fonction de transfert actuelle.

Pour le premier problème la solution présentée passe par des structures dites mutables et une fonction appelée lors des commutations des interrupteurs pour modifier la fonction de transfert utilisée.

Pour le second problème nous avons utilisé une boite à outils spécifique de Julia : DifferentialEquations [4]. Cette boite à outils est programmée en Julia, très performante, très simplement configurable.

Pour utiliser cette boite à outils, il suffit de mettre le code suivant :

```
using DifferentialEquations
```

3.1. *Prise en compte des commutations avec Julia*

La solution retenue pour cette prise en compte se compose de deux parties : d'abord une structure de variables pour garder en mémoire l'états des interrupteurs, puis l'usage d'une fonction pour piloter le changement d'état des interrupteurs.

Pour la définition de la variable, le type de commande choisit pour notre problème nécessite de connaitre à chaque instant :

- Le temps de la prochaine periode de PWM, que nous appellerons time_next_PWM
- Le temps de la prochaine commutation, que nous appellerons time_PWM_chg ;
- La valeur du rapport cyclique, que nous appellerons value ;
- L'état actuelle des interrupteurs du système, que nous appellerons state.

Pour qu'une modification de cette variable dans les fonctions appelant cette structure soit possible, Julia permet l'usage de structure dites mutables. Le code de génération de cette structure est :

```
mutable struct PWM_struct
  time_next_PWM
  time_PWM_chg
  value
 state
end
```

Nous pouvons maintenant créer la variable PWM1 qui utilise cette structure :

```
PWM1 = PWM_struct(periode_PWM, periode_PWM*PWM_init,PWM_init,Init_state)
```

Avec periode_PWM la periode de la MLI, PWM_init la valeur initiale du rapport cyclique et Init_state l'état initial des interrupteurs.

La modification périodique de ces valeurs est rendue possible par une possibilité de la boite à outils DifferentialEquations : l'appel à un temps définie d'une fonction préalablement définie avec l'appel IterativeCallback. Pour cela il faut créer deux fonctions : la première, appelée ici PWM, retourne le temps du prochain appel, la seconde, appelée ici effect_state!, réalise les modifications attendues :

```
function PWM(integrator)
  return(PWM1.time_PWM_chg)
end

function affect_state!(integrator)
  if PWM1.state == 2
    PWM1.time_PWM_chg = PWM1.time_next_PWM + periode_PWM*PWM1.value
    PWM1.time_next_PWM += periode_PWM
    PWM1.state = 1
   elseif PWM1.state == 1
    PWM1.time_PWM_chg = PWM1.time_next_PWM
    PWM1.state = 2
   end
end
```

Il faut enfin définir pour le solveur l'usage de ces fonctions, ce qui est réalisé par les appels suivants :

```
cb = IterativeCallback(PWM, affect_state!,)
cbs = CallbackSet(cb)
```

3.2. Résolution des équations différentielles avec Julia

Pour définir une équation différentielle, il faut créer une fonction qui retourne la valeur des dérivés en fonction des autres paramètres :

```
f(x,p,t) = A[PWM1.state]*x + B[PWM1.state].*Uin
```

Avec A et B les matrices présentées dans l'équation (2). La variable PWM1.state représente l'état des interrupteurs, bloqués ou passants.

L'appel initial de la fonction permet de déclarer l'état initial des variables d'état X0. Pour une comparaison pertinente avec le simulateur de référence Matlab/Simulink/Simscape Electrical, l'état initial du système est défini à 0 pour analyser la période transitoire du convertisseur.

Il faut aussi définir le temps de simulation, créer le problème à résoudre avec la fonction ODEProblem spécifique au package Differential Equations, et d'appeler cette fonction avec la méthode solve :

```
X0 = (0.0, 0.0)
Tspan = (0.0, 0.1)
prob = ODEProblem(f,x0,tspan)
solution = solve(prob, callback = cbs)
```

Il ne reste plus qu'à afficher le résultat. Une boite à outils est disponible pour cela : Plots. Pour afficher la tension du condensateur et le courant de l'inductance est le suivant :

```
using Plots
plot(solution, vars = 2, label = "Tension du condensateur")
plot!(solution, vars = 1, label = "Courant de l'inductance")
```

4. RÉSULTATS

Les paramètres de la simulation sont indiqués dans le tableau suivant :

Tableau 1. Paramètre de la simulation

	U_e	L	C	R_S	f
Valeur	10 V	1mH	1mF	10Ω	10kHz

La figure (2) compare le résultat de la simulation avec Julia et de la simulation avec Matlab Simulink et la toolbox Simscape Electrical.

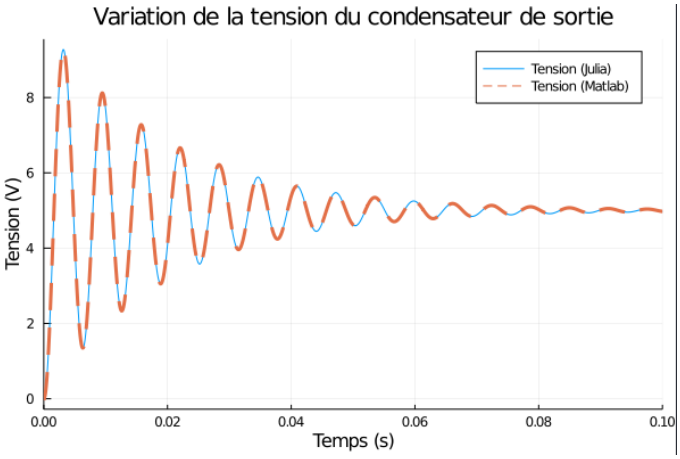

Fig. 2. Comparaison entre la simulation avec Julia et avec Simscape Electrical avec différentes valeurs de α.

Les deux résultats sont identiques, Ce résultat confirme que la simulation sous Julia est efficace pour réaliser ces simulations, avec l'avantage de la gratuité pour Julia.

Reste maintenant la question de la performance du calcul. Pour cela, une boite à outils est disponible : BenchmarkTools. Son usage est simple, il suffit d'indiquer l'utilisation de la boite à outils, puis d'appeler la fonction dont on veut connaitre la performance avec l'outil :

```
using BenchmarkTools
@benchmark solve_EdP solve(prob, callback = cbs)
```

Cet outil permet d'appeler la fonction un grand nombre de fois et de donner comme résultat : le temps de calcul minimal, le temps maximal, le temps moyen et le temps median.

Le résultat de la simulation avec Julia, sur un ordinateur ayant un CPU AMD Ryzen 7 2700 est donné dans le tableau 2.

Il est possible d'appeler le compilateur Matlab avec Julia, et d'utiliser le même outil de comparaison de temps de calcul entre Matlab 2020a et Julia. L'utilisation d'un appel de fonction entraine une légère hausse du temps de calcul, évalué avec l'outil Benchmark comme temps median 82 µs, comme temps maximal 452 µs et comme temps minimal 70 µs. Comme l'indique le tableau 2, ce temps d'appel est négligeable vis-à-vis de la durée de calcul de simulation.

La simulation sous Matlab est de plus faite avec une pré-compilation et un fichier de caches permettant d'accélérer le calcul.

Tableau 2. Benchmark du simulateur sous Julia et Matlab

Durée de calcul	**Minimal (ms)**	**Median (ms)**	**Maximal (ms)**
Julia	0,350	7,4	24
Matlab/Simulink/ Simscape Electrical	942	979	990

Le ratio de temps entre Matlab et Julia est, pour la valeur médiane, de plus de 130 en faveur de Julia. Ce résultat est très positif.

La durée du calcul de la simulation avec Julia est, même pour le cas Maximal, très supérieur au besoin du temps réel : la durée simulée est de 0.1 s pour une durée de calcul de 0.024 s. Certains auteurs commencent d'ailleurs à utiliser le langage Julia pour du contrôle temps réel de système robotique[5].

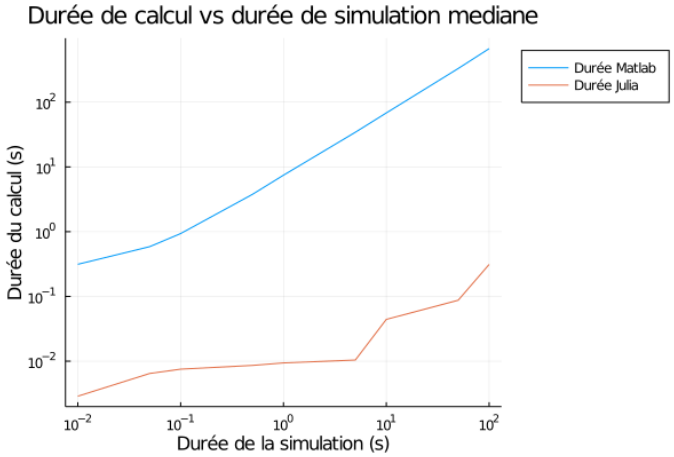

Fig. 3. Courbe log/log du temps médian de calcul avec Matlab et Julia en fonction de la durée de la simulation.

La figure 3 confirme que le temps de calcul est bien meilleur quelque soit la durée de la simulation, de 0.01s à 100s. La courbe de durée de simulation sous Julia n'est pas régulière, mais ce point n'est pas encore éclairci.

5. Conclusions

L'usage d'un langage de programmation haut niveau permettant une exécution simple en calcul parallèle ou en calcul distribué permet de gagner du temps lors de recherche nécessitant un nombre important de simulation multiple comme un algorithme génétique ou tout autre optimisation.

Le résultat de la simulation d'un convertisseur d'électronique de puissance est très encourageant : simple à mettre en place, cette simulation est très efficace en temps de calcul sans l'usage des facultés supplémentaire de Julia : la mise en œuvre simplifiée d'un calcul parallèle, distribué ou sur GPU. La boite à outils DifferentialEquations permet ce type de calcul simplement, mais pour être utile il faut un nombre de variable d'état beaucoup plus important pour compenser le temps de communication entre les cœurs ou avec le GPU.

Pour un usage facilité du système, il faut maintenant développer un ensemble de fonction permettant de définir simplement un convertisseur complexe sans avoir à définir l'ensemble des matrices de transfert possible.

Il faudra par ailleurs ajouter un type de composant plus complexe à programmer : les composants non pilotés comme les diodes.

6. References

[1] J. Bezanson, S. Karpinski, V. B. Shah, et A. Edelman, « Julia: A Fast Dynamic Language for Technical Computing », *ArXiv12095145 Cs*, sept. 2012, Consulté le: oct. 06, 2020. [En ligne]. Disponible sur: http://arxiv.org/abs/1209.5145.

[2] D. Jacob, *Electronique de puissance Principes, fonctionnement, dimensionnement - Cours et problèmes résolus*, Eyrolles. 2008.

[3] M. Lubin et I. Dunning, « Computing in Operations Research Using Julia », *Inf. J. Comput.*, vol. 27, nº 2, p. 238-248, mars 2015, doi: 10.1287/ijoc.2014.0623.

[4] C. Rackauckas et Q. Nie, « DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia », *J. Open Res. Softw.*, vol. 5, nº 1, p. 15, mai 2017, doi: 10.5334/jors.151.

[5] T. Koolen et R. Deits, « Julia for robotics: simulation and real-time control in a high-level programming language », in *2019 International Conference on Robotics and Automation (ICRA)*, mai 2019, p. 604-611, doi: 10.1109/ICRA.2019.8793875.