HEIG VD
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

# Génération de trajectoires à la demande pour le contrôleur du Beam Wire-Scanner du CERN

Département TIC

Filière Informatique et systèmes de communication

Orientation Systèmes informatiques embarqués

Clément Dieperink

10 août 2023

CERN-THESIS-2023-331
//2023

Sujet proposé par :
Jonathan Emery (CERN)

Supervisé par :
Prof. Yann Thoma (HEIG-VD)

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

# Préambule

Ce travail de Bachelor (ci-après **TB**) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'École.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD
Le Chef du Département

Yverdon-les-Bains, le 10 août 2023

# Authentification

Je soussigné, Clément Dieperink, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Clément Dieperink

Yverdon-les-Bains, le 10 août 2023

# Résumé

Le CERN utilise des scanners dans ces accélérateurs de particules pour mesurer la position des faisceaux de particules. Ces scanners, appelés *Beam Wire-Scanner*, sont composés d'un moteur auquel est accroché un fil très fin. En se déplaçant rapidement au travers du faisceau, ce fil interagit avec celui-ci et produit des particules secondaires qui sont ensuite détectées par un scintillateur. La taille transversale et la position du faisceau sont alors déterminées. Le fil étant fragile, les mouvements doivent être doux pour ne pas le casser.

Plusieurs vitesses d'interaction avec le faisceau et distances de déplacement parcourues sont utilisées pour les mesures. Différents systèmes sont également utilisés, changeant les paramètres liés au courant électrique nécessaire du moteur.

Les trajectoires, de type *S-Curve*, suivies par les moteurs, sont générées en avances dans des tables de valeur qui sont ensuite compilées avec le projet. Ce processus demande beaucoup de ressource du circuit logique programmable limitant le nombre de trajectoires et de système pouvant être contrôlé par un seul binaire. Le projet doit également être compilé et déployé pour chaque changement dans une trajectoire.

Ce projet répond à cette problématique en proposant une solution permettant de calculer les trajectoires directement sur le contrôleur du moteur. Cette solution offre la possibilité de paramétrer les trajectoires à la volée et sans limitation avec un seul binaire, simplifiant ainsi la maintenance et l'utilisation des systèmes.

Pour cela, l'algorithme de génération a été optimisé et implémenté avec le langage de description matériel VHDL.

Un banc de test automatique permettant de valider la description a également été développé. Des vérifications supplémentaires ont été ajoutées lors du calcul des trajectoires pour garantir de ne pas dépasser des limites physiques pouvant abîmer le système.

Le résultat final est fonctionnel et pourra être intégré dans le projet global de ces scanners.

# Table des matières

# Table des figures

# Liste des tableaux

# Liste des codes sources

# Chapitre 1

# Introduction

## *1.1 Contexte*

Ce travail est réalisé en collaboration avec le groupe d'instrumentation des faisceaux de l'Organisation Européenne pour la Recherche Nucléaire (CERN). Ce groupe est responsable de rechercher, développer, construire et maintenir les instruments de mesures des faisceaux de particules dans les accélérateurs du CERN.

Ce projet concerne principalement le *Beam Wire-Scanner* (BWS) qui est un instrument pour mesurer la taille transversale du faisceau de particules. Ce dispositif mécatronique est constitué d'un fil très fin qui traverse rapidement le faisceau de particules dans le vide. Cette interaction produit des particules secondaires qui sont détectées par un scintillateur externe permettant ainsi de mesurer la taille et position du faisceau dans l'accélérateur.

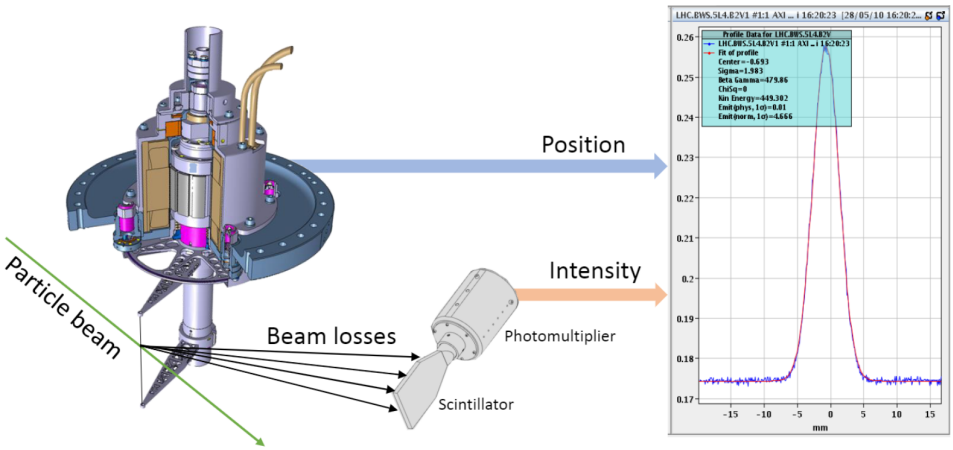


**Figure 1.1** – *Schéma du fonctionnement du* Beam Wire-Scanner

Le fil est monté au bout d'une fourche déplacée par un moteur brushless contrôlé par une FPGA. Ce système a besoin d'un mouvement fluide pour différentes vitesses et longueurs de course, tout en garantissant un degré élevé de confiance dans la qualité de la trajectoire prédéfinie. Étant donné que plusieurs paramètres physiques influencent le comportement de ces systèmes, il est difficile d'appliquer une trajectoire identique à tous les instruments. Ce projet se concentre sur un mouvement rotatif, mais celui-ci peut également être linéaire.

## 1.2 Objectifs

Actuellement, les trajectoires sont pré-calculées à l'aide d'un algorithme implémenté en Matlab et Python pour respectivement les mouvements rotatifs et linéaires. Plusieurs tables sont donc générées et intégrées à la synthèse dans le projet de la FPGA comme mémoire en lecture seule. Cette méthode n'est pas optimale, car elle utilise beaucoup de ressources de la FPGA limitant le nombre de trajectoires différentes possible et le nombre de scanners contrôlables avec le même binaire. Il faut également re-synthétiser tout le projet pour changer les paramètres d'une trajectoire.

L'objectif est donc de générer les trajectoires directement dans la FPGA. Cela permettra de n'avoir qu'un seul binaire pour contrôler tous les systèmes avec des paramètres différents.

## 1.3 Travail effectué

Une première partie d'analyse a été réalisée afin de déterminer l'algorithme qui sera utilisé pour générer les trajectoires. Deux algorithmes sont présentés, le premier est actuellement utilisé et le second est une alternative qui pourrait être utilisée dans le futur.

L'algorithme actuel est choisi pour la suite du projet, car les trajectoires générées ont déjà pu être validées sur un moteur réel. Une optimisation de celui-ci est proposée afin de réduire le nombre d'opérations nécessaires pour le calcul. Finalement, les équations sont discrétisées et adaptées pour une implémentation en VHDL.

Le type des données utilisées est également déterminé afin d'avoir une précision suffisante tout en limitant l'impact sur les ressources de la FPGA.

Après cela, des schémas sont proposés. Ceux-ci montrent le fonctionnement de l'implémentation de l'algorithme sur la FPGA, sans être trop détaillé.

Les blocs sont ensuite implémentés en VHDL. Les descriptions ont été écrites pour être les plus génériques possible. Celles-ci sont séparées en deux fichiers pour le générateur et la relecture et un dernier faisant le lien entre les deux autres.

Finalement, plusieurs bancs de test sont réalisés afin de vérifier le bon fonctionnement de l'implémentation et de comparer les résultats avec une référence.

Une documentation simple pour l'utilisation du générateur est également réalisée et disponible en annexe A. Cette documentation a pour but de permettre l'utilisation du générateur sans avoir à lire le rapport complet.

## 1.4 Structure du document

Le chapitre **1 Introduction** présente le contexte (1.1) du projet, les objectifs (1.2) et le travail effectué (1.3).

Le chapitre **2 Algorithme de génération** présente les spécifications de l'algorithme (2.1), l'algorithme actuel et son optimisation (2.2), le second algorithme (2.3), les calculs pour le courant (2.4) et les adaptations des équations pour une implémentation en VHDL (2.5).

Le chapitre **3 Conception** présente les schémas des différents blocs du système de génération de trajectoire.

Le chapitre **4 Implémentation** présente l'implémentation des différents blocs en VHDL avec les détails sur les choix pris lors de l'écriture des descriptions. Les résultats de synthèses sont également présentés (4.4).

Le chapitre **5 Banc de test** présente les bancs de test réalisés pour vérifier le bon fonctionnement de l'implémentation et comparer les résultats avec une référence. Les résultats des tests sont également présentés (5.4).

Le chapitre **6 Conclusion** présente les conclusions du projet.

Les annexes présentent la documentation d'utilisation (A), la planification (B), la comparaison des algorithmes (C) et la comparaison des types de variables (D).

# Chapitre 2

# Algorithme de génération

Afin de générer des trajectoires correctes, un algorithme doit être choisi et adapté à une implémentation sur FPGA. Ce chapitre présente les spécifications à respecter ainsi que deux algorithmes, celui actuellement utilisé et un second qui pourrait être utilisé à l'avenir. La conversion des trajectoires en courant électrique nécessaire au moteur est également présenté.

Une optimisation pour l'implémentation de l'algorithme actuel est proposée. Celle-ci permet de grandement réduire le nombre d'opérations nécessaires pour générer les trajectoires. Les équations nécessaires sont discrétisées et adaptées pour une implémentation sur FPGA, le résultat final est ensuite comparé à la version actuelle et le type des données qui sera utilisé est discuté.

## *2.1 Spécification de l'algorithme*

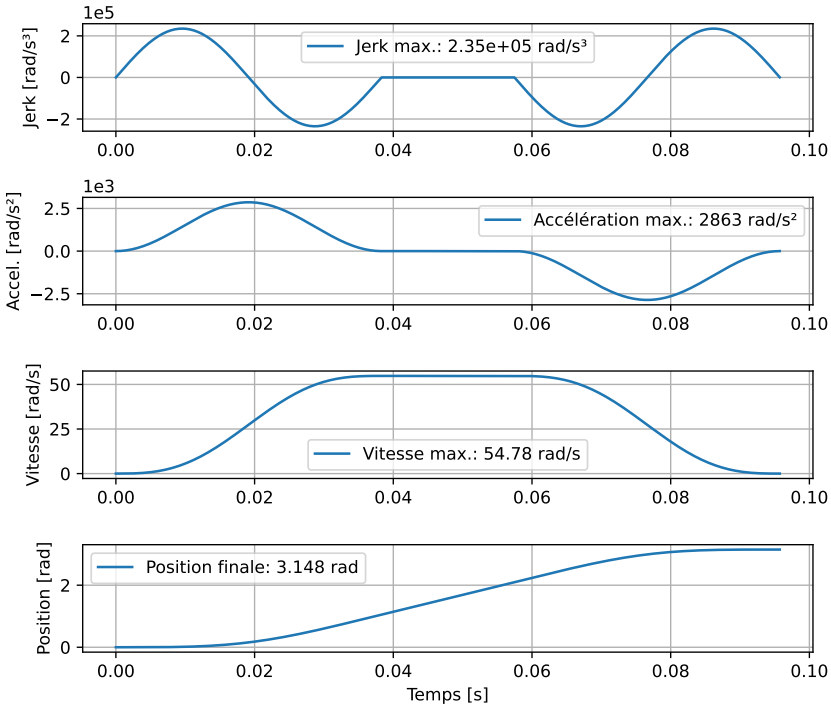


**Figure 2.1** – *Exemple de génération d'une trajectoire avec l'algorithme actuel*

Une trajectoire de type *S-Curve* doit être générée. Ce type de trajectoire est utilisé pour rendre les mouvements plus fluides et éviter les à-coups, ce qui est important pour ne pas endommager le fil du BWS. La figure 2.1 présente une trajectoire générée avec l'algorithme actuellement utilisé. Trois étapes distinctes sont présentes : l'accélération, la vitesse constante et la décélération.

Plusieurs valeurs sont présentes sur cette figure : la position, la vitesse angulaire, l'accélération et le jerk. Ce dernier est la dérivée de l'accélération et montre comment celle-ci varie dans le temps.

Pour le fonctionnement du BWS, il est nécessaire de calculer la position ($\theta(t)$ [rad]), la vitesse angulaire ($\omega(t)$ [rad s$^{-1}$]) ainsi que le courant nécessaire ($I_q(t)$ [A]) au moteur. L'accélération ($\alpha(t)$ [rad s$^{-2}$]) est nécessaire pour le calcul du courant et le jerk ($j(t)$ [rad s$^{-3}$]) n'est pas nécessaire, mais peut être utilisé pour la génération de la trajectoire. Ces valeurs seront utilisées par le reste du projet dans un contrôleur à boucle fermée pour s'assurer que le moteur suit correctement la trajectoire.

Le moteur fonctionne avec une *Pulse Width Modulation* (PWM). Les valeurs générées pour la trajectoire doivent donc être espacées dans le temps de façon à respecter la période de la PWM. La fréquence de la PWM est d'environ 16 kHz.

La génération de la trajectoire peut être faite en amont de son utilisation, aucune contrainte de temps n'est donc présente pour la génération.

La trajectoire est générée à partir de plusieurs paramètres d'entrée :

$\theta_{start}$ = Position de départ de la trajectoire [rad]
$\omega_{target}$ = Vitesse d'interaction avec le faisceau [rad s$^{-1}$]
$\theta_{acc}$ = Angle de déplacement pour l'accélération [rad]
$\theta_{dec}$ = Angle de déplacement pour la décélération [rad]
$\theta_{total}$ = Angle de déplacement total parcouru [rad]

La vitesse d'interaction avec le faisceau correspond à la vitesse à atteindre lorsqu'elle est constante. Les angles de déplacement pour l'accélération et la décélération correspondent à la distance parcourue par le moteur pour, respectivement, atteindre la vitesse d'interaction et s'arrêter. L'angle de déplacement total correspond à la distance totale parcourue par le moteur. Ces différentes valeurs peuvent être soit positives, soit négatives, mais doivent toutes être du même signe à l'exception de $\omega_{start}$. Ce signe indique le sens de rotation du moteur.

Le moteur devant faire des mouvements aller-retour, il a été demandé de, si possible, calculer les trajectoires dans les deux sens indépendamment l'une de l'autre. Ceci permet de s'assurer que les trajectoires soient correctes dans les deux sens. En effet, la position finale à l'aller ne sera pas exactement la position de départ au retour, dû à des imprécisions et au frein magnétique, qui sera discuté par la suite. Les directions aller et retour seront également référencées par les termes *in* et *out*.

D'autres paramètres sont nécessaires pour la conversion de la trajectoire en courant électrique. Ceux-ci sont présentés ci-dessous :

$J_{tot}$ = Inertie du système (moteur + charge) [kg m$^2$]
$\mu_{dyn}$ = Facteur de friction dynamique [N m s rad$^{-1}$]
$\tau_{sta}$ = Équivalent de torque de friction statique [N m]
$K_\tau$ = Constante de couple du moteur [N m A$^{-1}$]
$\tau_{mbs}(\theta)$ = Couple du frein magnétique en fonction de la position [N m]

L'inertie du système est la somme de l'inertie du moteur et de la charge nécessaire à déplacer. Le facteur de friction dynamique correspond à la friction en fonction de la vitesse du moteur lorsqu'il tourne. La constante de couple est utilisée pour convertir le couple en courant électrique.

Un frein magnétique permet de bloquer le moteur en dehors du faisceau lorsque le moteur est à l'arrêt. Ce frein produit un couple variant en fonction de la position du moteur qui doit être pris en compte lors du calcul du courant. Une *Look Up Table* (LUT) est utilisée pour obtenir ce couple et fait partie des paramètres modifiables en entrée. Ce frein n'est pas présent sur tous les moteurs, un paramètre supplémentaire permet de l'activer ou non.

Certaines limites physiques doivent être respectées pour éviter d'endommager le moteur ou le fil. La vitesse, l'accélération et le courant ne doivent donc pas dépasser une valeur absolue maximale. La position doit également rester dans une plage définie. Dans le cas où une de ces limites est dépassées, la trajectoire doit être indiquée comme invalide et ne doit donc pas être utilisée. Une erreur est également indiquée si les paramètres d'entrée $\omega_{target}$, $\theta_{acc}$, $\theta_{dec}$, $\theta_{total}$ et $K_\tau$ sont égaux à zéro, car cela peut générer des divisions par zéro ou des valeurs infinies lors de la génération.

## 2.2 Algorithme de génération actuel

L'algorithme actuellement utilisé est présenté dans [Eme+19]. Pour générer les étapes d'accélération et de décélération, cette version se base sur un jerk sinusoïdal qui est intégré trois fois pour obtenir l'accélération, la vitesse et la position du moteur lors des phases d'accélération et de freinage. Les trajectoires générées par cet algorithme ont pu être testées et validées sur le BWS afin de s'assurer qu'elles ne créent pas de vibrations trop importantes.

La figure 2.1 montre les différentes valeurs d'une trajectoire générée avec cet algorithme et les paramètres suivants :

$$
\begin{aligned}
\omega_{target} &= 55\,\mathrm{rad\,s^{-1}} \\
\theta_{acc} &= \tfrac{\pi}{3}\ \mathrm{rad} \\
\theta_{dec} &= \tfrac{\pi}{3}\ \mathrm{rad} \\
\theta_{total} &= \pi\ \mathrm{rad}
\end{aligned}
$$

### 2.2.1 Implémentation actuelle

L'implémentation actuelle de l'algorithme pour la génération des mouvements rotatifs est faite dans un script Matlab. Ce script crée des tables de valeurs pour les différentes trajectoires qui sont ensuite ajoutées à la synthèse de la description du BWS.

#### *Étapes d'accélération et de décélération*

Les étapes d'accélération et de décélération sont générées de manières similaires. L'implémentation actuelle recherche la durée nécessaire et le jerk maximal pour atteindre la vitesse cible sur la distance voulue à l'aide de deux boucles imbriquées pour tester plusieurs durées et jerk différents. Ces boucles calculent pour chaque itération l'étape complète avec les trois intégrales jusqu'à ce que la distance parcourue et la vitesse finale soient celles voulues. Ceci implique que beaucoup d'opération

sont effectuées pour trouver les valeurs optimales de temps et de jerk. Une réécriture telle quelle de cet algorithme en VHDL serait très inefficace.

Afin de limiter le temps de recherche, les durées et jerks testés sont limités à des plages définies. Un pas est également défini entre chaque valeur testée. Cela a pour effet que la vitesse atteinte ne va que très rarement être exactement celle voulue, ainsi une plage proche de la vitesse cible est définie pour laquelle celle-ci est considérée comme atteinte. La distance à parcourir est considérée comme atteinte lorsque la position finale est supérieure à la position voulue, ce qui implique que celle-ci sera toujours légèrement dépassée. Les trajectoires générées ayant été validées sur le BWS, ces imprécisions ne semblent pas poser de problèmes.

#### *Étape à vitesse constante*

L'étape à vitesse constante est plus simple à générer. La position est calculée en intégrant la vitesse sur le temps totale de l'étape qui peut être calculé à l'aide de la distance à parcourir et de la vitesse atteinte après l'accélération.

### *2.2.2 Optimisation de l'algorithme*

Lors des étapes d'accélération et de décélération, le jerk est toujours généré à partir d'une sinusoïde de la forme :

$$\begin{aligned} t &\in [0; T_{acc}] \\ j(t) &= \sin(\frac{2\pi}{T_{acc}} \cdot t) \end{aligned} \tag{2.1}$$

Avec :

$T_{acc}$ = Temps nécessaire pour atteindre la vitesse cible [s]
$t$ = Temps courant de l'étape [s]
$j(t)$ = Jerk à l'instant $t$ [rad s$^{-3}$]

La valeur de $\frac{2\pi}{T_{acc}} \cdot t$ sera référencées par $x$ dans les prochaines équations.

$$x(t) = \frac{2\pi}{T_{acc}} \cdot t \tag{2.2}$$

Cette sinusoïde est ensuite intégrée trois fois pour obtenir l'accélération, la vitesse et la position. Les équations suivantes sont obtenues :

$$\alpha(x) = \int_0^x j(x)dx = 1 - \cos(x)$$

$$\omega(x) = \int_0^x a(x)dx = x - \sin(x) \tag{2.3}$$

$$\theta(x) = \int_0^x v(x)dx = \frac{x^2}{2} + \cos(x) - 1 \tag{2.4}$$

Avec :

$\alpha(t)$ = Accélération à l'instant $t$ [rad s$^{-2}$]
$\omega(t)$ = Vitesse à l'instant $t$ [rad s$^{-1}$]
$\theta(t)$ = Position à l'instant $t$ [rad]

Sans autres modifications, ces fonctions donnent la trajectoire d'accélération avec une amplitude de 1 pour le jerk, comme le montre la figure 2.2.

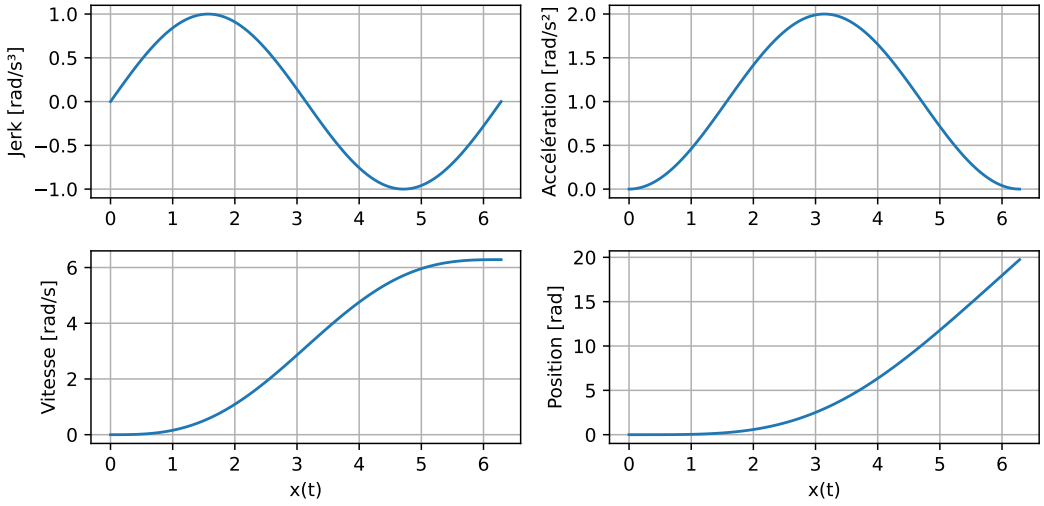


**Figure 2.2** – *Génération de l'étape d'accélération avec les fonctions*

Afin de simplifier l'implémentation sur FPGA, seule la vitesse sera calculée directement à l'aide de l'équation 2.3. La position est alors obtenue en intégrant la vitesse et l'accélération en la dérivant.

Connaissant la vitesse à atteindre, l'équation de la vitesse peut être multipliée par le facteur $\frac{\omega_{target}}{\omega(2\pi)}$ pour obtenir la vitesse voulue :

$$\omega_2(x) = \omega(x) \cdot \frac{\omega_{target}}{\omega(2\pi)} = (x - \sin(x)) \cdot \frac{\omega_{target}}{2\pi} \tag{2.5}$$

Avec la vitesse moyenne ($\overline{\omega_2(x)}$) sur la durée complète d'accélération, il est possible de calculer la distance parcourue :

$$\overline{\omega_2(x)} = \frac{\int_0^{2\pi} \omega_2(x)dx}{2\pi} = \frac{\omega_{target}}{2} \tag{2.6}$$

$$\theta_{acc} = T_{acc} \cdot \overline{\omega_2(x)} \text{ pour } x \in [0; 2\pi] \tag{2.7}$$

La distance parcourue étant un paramètre d'entrée de la trajectoire, il est possible de calculer la durée $T_{acc}$ avec la relation suivante :

$$T_{acc} = \frac{2\theta_{acc}}{\omega_{target}} \tag{2.8}$$

La décélération est générée avec le même principe en parcourant la sinusoïde à l'envers. La figure 2.3 montre la trajectoire pour l'accélération et le freinage obtenue à partir de l'équation 2.5 avec une vitesse cible de $55\,\mathrm{rad\,s^{-1}}$ et une distance de $\frac{\pi}{3}$ rad. L'accélération, la position et le jerk sont obtenus en dérivant et intégrant la vitesse. Il est à noter que les positions pour la partie de décélération devront être cumulées avec la position finale obtenue lors de l'étape à vitesse constante.

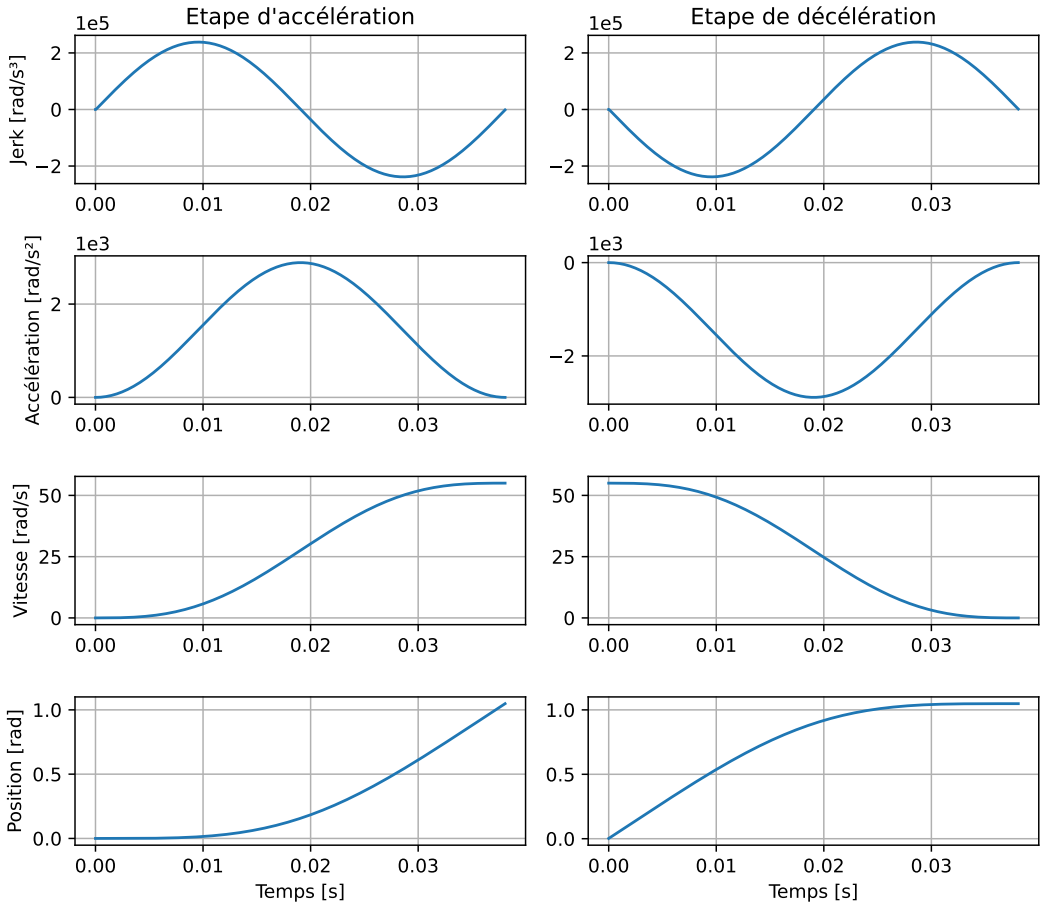


**Figure 2.3** – *Génération des étapes d'accélération et de décélération avec les facteurs de vitesse et de temps*

## 2.3 Second algorithme proposé

Un second algorithme est en cours d'évaluation par le groupe d'instrumentation des faisceaux du CERN et est présenté dans [BM21]. Celui-ci permet de générer une trajectoire *S-Curve* prenant le moins de temps possible tout en respectant les limitations cinématiques du système. Il est également capable de supprimer les vibrations dues au mouvement dans un système résonant.

Cet algorithme génère la trajectoire en filtrant une entrée impulsionnelle dans une cascade de $n$ filtres dynamiques, où $n$ correspond au degré de la dérivée la plus élevée utilisée pour générer la trajectoire.

La figure 2.4 montre la génération de la trajectoire avec le second algorithme pour des valeurs proches de celles obtenues dans la figure 2.1. La trajectoire finale générée ressemble à celle de la figure 2.1, mais les changements du jerk sont plus brutaux. Il

est possible de rendre le jerk plus doux en augmentant le nombre de filtres utilisés pour générer la trajectoire.

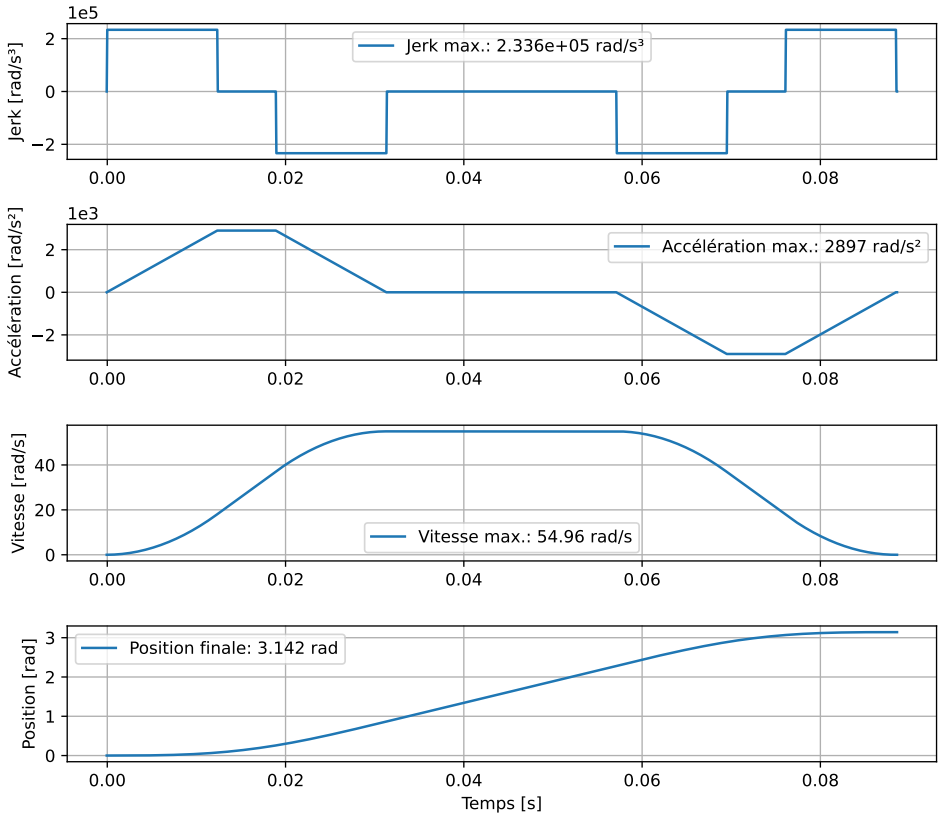


**Figure 2.4** – *Génération d'une trajectoire avec le second algorithme*

Cet algorithme prend en entrée la distance totale à parcourir ainsi que les valeurs maximales pour chaque dérivée de la position (vitesse, accélération, jerk, ...) jusqu'au degré voulu. Les durées nécessaires pour les différentes étapes sont calculées à partir de ces valeurs. Ces paramètres ne sont donc pas tout à fait ceux voulus par la spécification, il manque la notion de distance parcourue pour les étapes d'accélération et de décélération. L'article [BM21] présente une méthode pour calculer les durées optimales pour minimiser le temps de parcours et les vibrations.

L'avantage de cet algorithme est qu'il permet de générer la trajectoire de façon itérative par pas de la période de la PWM avec seulement quelques opérations par itération et degré de la dérivée. L'équation discrète utilisée pour générer la trajectoire est présentée ci-dessous et provient de [BM12] :

$$q_i(k) = q_i(k-1) + \frac{1}{N_i}(q_{i-1}(k) - q_{i-1}(k-N_i)) \tag{2.9}$$

$$q_0(k) = \begin{cases} \theta_{final} & \text{pour } k = 0, 1, 2, \dots \\ 0 & \text{pour } k < 0 \end{cases} \tag{2.10}$$

$$N_i = \frac{T_i}{T_s} \tag{2.11}$$

Avec :

$q_n(k)$ = Position de la trajectoire d'ordre $n$ à l'instant $k$ [rad]
$N_i$ = Nombre d'itérations pour la dérivée $i$
$\theta_{final}$ = Position finale à atteindre [rad]
$T_i$ = Durée de la réponse impulsionnelle pour chaque filtre [s]
$T_s$ = Période d'échantillonnage (PWM) [s]

La vitesse et l'accélération peuvent être obtenues en dérivant la position puis la vitesse. Les valeurs étant discrète, la dérivée est simplement calculée en soustrayant la position actuelle de la précédente et en divisant le résultat par la période de la PWM.

Contrairement à l'algorithme actuel, celui-ci ne va pas forcément générer une trajectoire où la vitesse constante est atteinte. Il ne permet pas non plus de s'assurer de la distance parcourue par les différentes étapes. Cependant, il permet de générer des trajectoires dont chaque dérivée est limitée à une valeur maximale. Il n'a pas encore pu être testé sur le système réel et n'est donc pas retenu pour ce projet. Il est cependant intéressant de le présenter, car il pourrait être utilisé dans le futur en fonction des résultats obtenus.

## 2.4 Calcul du courant

Le courant électrique nécessaire au moteur est calculé à partir de l'accélération, de la vitesse et des paramètres présentés à la section 2.1. Les équations 4a, 4b et 4c de [Eme+19] sont utilisées pour calculer le courant électrique nécessaire au moteur. Ces équations sont présentées ci-après :

$$\tau_{brake}(t) = \tau_{mbs}(\theta(t)) \tag{2.12}$$

$$\tau(t) = \alpha(t) \cdot J_{tot} + \omega(t) \cdot \mu_{dyn} + \tau_{brake}(t) + \tau_{sta} \tag{2.13}$$

$$I_q(t) = \frac{\tau(t)}{K_\tau} \tag{2.14}$$

Le couple du frein magnétique $\tau_{mbs}$ est obtenu à partir d'une table de valeur mesurée. Afin d'obtenir une valeur le plus proche de la valeur réelle, une interpolation est faite sur les valeurs de la table. Actuellement, une *spline* cubique est utilisée.

Selon l'algorithme présenté sur la page correspondante de Wikipédia [1], l'équation suivante peut être utilisée pour calculer l'interpolation :

$$S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \tag{2.15}$$

Quatre coefficients sont nécessaires pour chaque intervalle $j$ des valeurs contenues dans la table. Le nombre d'opérations pour générer ces coefficients n'est pas négligeable. Cependant, ceux-ci ne doivent être calculés qu'une seule fois et il est possible de les stocker dans la mémoire. La génération des coefficients prendra également de la place sur la FPGA. En fonction des besoins, plusieurs options sont envisageables pour en limiter l'impact :

1. https://en.wikipedia.org/wiki/Spline_(mathematics)

1. Calculer les coefficients à la compilation
2. Rendre les coefficients paramétrables au lieu de la table des valeurs
3. Utiliser une interpolation linéaire

Les deux premières options ont l'avantage de ne pas nécessiter de calculer les valeurs dans la FPGA. Cependant, la première nécessite la recompilation du code pour changer les valeurs de la table et la deuxième fait perdre à la description la responsabilité totale sur la génération des trajectoires. La troisième option permet de réduire le nombre de coefficients à calculer et la complexité du calcul de l'interpolation. Ceci se fait au détriment de la précision du résultat.

Le nombre d'éléments dans la LUT est fixe à la compilation et les valeurs sont espacées d'un nombre constant de degré entre elles, ce qui permet de simplifier le calcul de l'interpolation.

### *2.4.1 Comparaison des splines cubique et linéaire*

La figure 2.5 présente la différence entre l'interpolation cubique et linéaire avec la table de valeurs données. La différence maximale entre les deux interpolations est d'environ 0.03 N m et se situe aux alentours de $\frac{\pi}{2}$ rad.

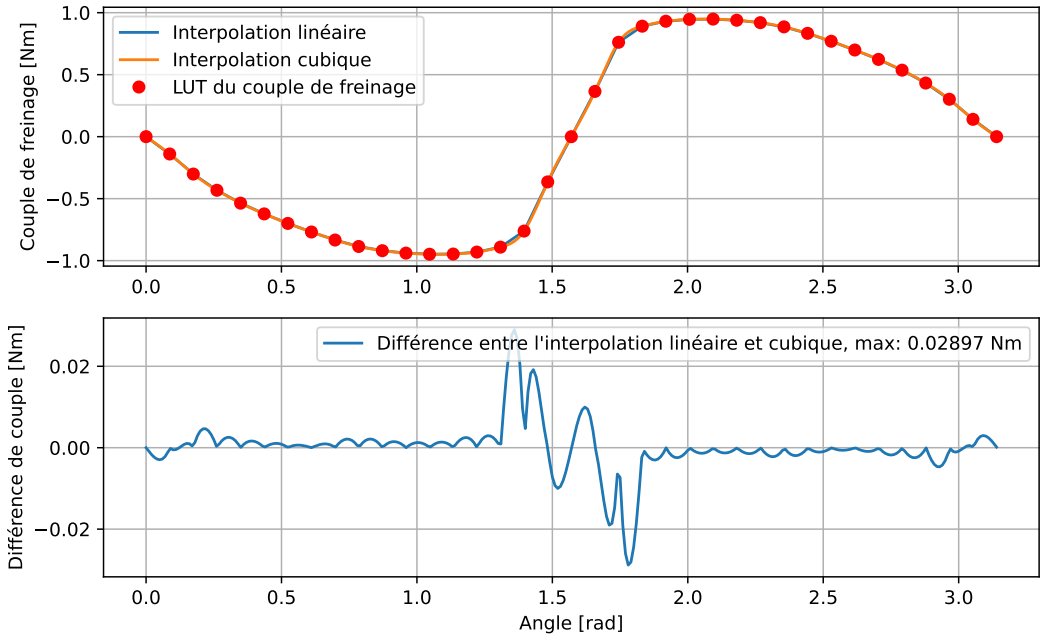


**Figure 2.5** – *Comparaison des splines cubique et linéaire*

La figure 2.6 présente la différence entre les deux interpolations sur le courant électrique nécessaire au moteur avec des vitesses cibles différentes. Dans les deux cas, la différence maximale est d'environ 0.016A. Cette différence n'étant pas significative l'interpolation linéaire est retenue.

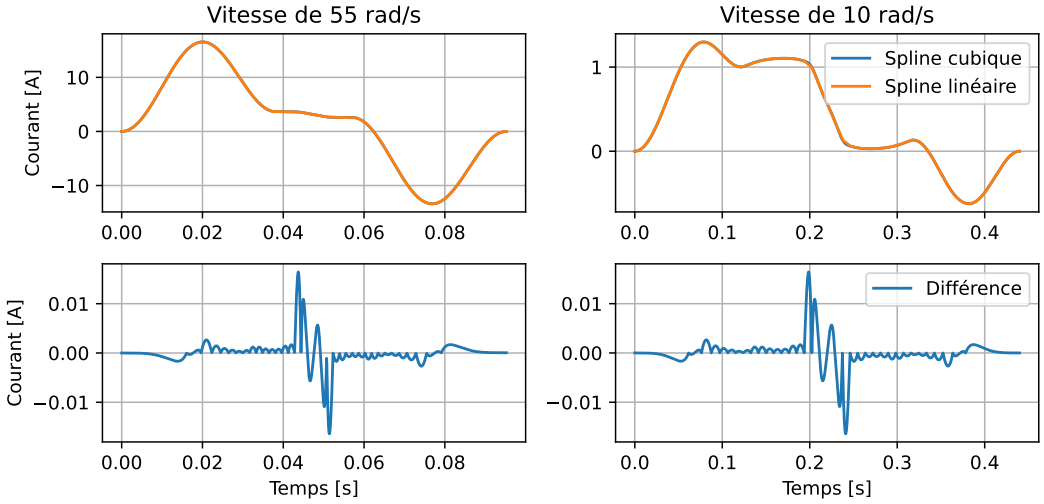


**Figure 2.6** – *Comparaison du courant avec les splines cubiques et linéaires avec deux vitesses*

## 2.5 Adaptation des équations pour la FPGA

### 2.5.1 Génération de la trajectoire

La génération de la trajectoire doit respecter la fréquence de la PWM entre chaque échantillon. Ainsi, les équations utilisées doivent être discrétisées. De plus, le calcul peut se faire de manière itérative, il est donc possible de réutiliser les valeurs précédentes. La position et l'accélération sont en tout temps égales à l'intégrale et la dérivée de la vitesse ce qui permet d'effectuer les mêmes calculs pour les trois étapes de la trajectoire.

$$k = 0, 1, 2, \ldots$$
$$\theta(k) = \theta(k-1) + \omega(k) \cdot T_{pwm} \tag{2.16}$$
$$\alpha(k) = (\omega(k) - \omega(k-1)) \cdot f_{pwm} \tag{2.17}$$

Avec :

$k$ = Indice de l'échantillon
$\theta(0)$ = $\theta_{start}$
$\alpha(0)$ = $0\,\mathrm{rad\,s^{-2}}$
$T_{pwm}$ = Période de la PWM [s]
$f_{pwm}$ = Fréquence de la PWM [Hz]

La vitesse quant à elle est calculée différemment pour chaque étape de la trajectoire. Lors de l'accélération et du freinage, l'équation 2.5 est utilisée et le reste du temps, la vitesse est constante.

La version discrétisée de l'équation 2.5 est la suivante :

$$k = 0, 1, 2, \ldots$$
$$x(k) = x(k-1) + incr \tag{2.18}$$
$$\omega(k) = (x(k) - \sin(x(k))) \cdot \frac{\omega_{target}}{2\pi} \tag{2.19}$$

Avec :

$k$ = Indice de l'échantillon de l'étape
$x_{acc}(0) = 0$
$x_{dec}(0) = 2\pi$
$incr_{acc}$ = Incrément de $x$ entre chaque échantillon durant l'accélération
$incr_{dec}$ = Incrément de $x$ entre chaque échantillon durant la décélération

La fonction 2.18 provient de l'équation 2.2 qui est rendue discrète et itérative.

Afin de n'avoir qu'une seule addition pour obtenir $x(k)$, les incréments sont des valeurs constantes pré-calculées à partir des paramètres d'entrée de la trajectoire :

$$incr_{acc} = \frac{2\pi T_{pwm}}{T_{acc}} = \frac{\pi T_{pwm}\omega_{target}}{\theta_{acc}} \tag{2.20}$$

$$incr_{dec} = -\frac{2\pi T_{pwm}}{T_{dec}} = -\frac{\pi T_{pwm}\omega_{target}}{\theta_{dec}} \tag{2.21}$$

Les étapes d'accélération et de décélération peuvent ainsi être combinées en un seul bloc avec les valeurs $x(0)$ et $incr$ appropriées. La constante $\frac{\omega_{target}}{2\pi}$ peut également être pré-calculée pour ne pas avoir à faire la division à chaque échantillon.

Afin de générer les trajectoires *out*, la solution choisie est d'inverser les signes des angles et vitesse des paramètres d'entrée de la trajectoire *in*. Cela permet de réutiliser les mêmes équations pour les deux directions. Les relations entre les paramètres d'entrée pour les deux directions sont les suivantes :

$\theta_{start,out} = \theta_{start,in} + \theta_{total,in}$
$\omega_{target,out} = -\omega_{target,in}$
$\theta_{acc,out} = -\theta_{dec,in}$
$\theta_{dec,out} = -\theta_{acc,in}$
$\theta_{total,out} = -\theta_{total,in}$

Les conditions pour passer d'une étape de vitesse à l'autre sont les suivantes :

| | |
|---|---|
| $x_{acc}(k) > 2\pi$ | accélération $\rightarrow$ constante |
| $\theta(k) > \theta_{start} + \theta_{total} - \theta_{dec}$ | constante $\rightarrow$ décélération ($in$) |
| $\theta(k) < \theta_{start} + \theta_{total} - \theta_{dec}$ | constante $\rightarrow$ décélération ($out$) |
| $x_{dec}(k) < 0$ | décélération $\rightarrow$ fin |

Cette nouvelle version de l'algorithme étant itérative, une erreur peut s'accumuler due à la précision des calculs. L'erreur cumulée peut être réduite en choisissant une précision suffisante sur les nombres utilisés. La section 2.5.4 présente le choix du type des variables utilisées pour les calculs afin d'avoir une précision suffisante.

### 2.5.2 Conversion du courant

Le calcul du courant nécessite l'équation 2.14. Celle-ci est déjà utilisable en l'état de façon discrète et itérative. Il est cependant possible de pré-calculer la valeur de $\frac{1}{K_\tau}$ afin de ne pas avoir à faire la division à chaque échantillon.

La fonction $\tau_{mbs}(\theta)$ utilisée dans cette équation effectue une interpolation linéaire à partir d'une table de valeur. La forme suivante de l'interpolation est utilisée :

$$\tau_{mbs}(\theta) = \tau_a + (\theta - \theta_a)\frac{\tau_b - \tau_a}{\theta_b - \theta_a} \tag{2.22}$$

Les valeurs $\theta_a$ et $\theta_b$ sont les angles de la table de valeurs les plus proches de $\theta$. $\tau_a$ et $\tau_b$ correspondent au torque associé à $\theta_a$ et $\theta_b$. Les valeurs de la table étant espacée d'un nombre constant de degrés, il est possible de pré-calculer le coefficient $\frac{1}{\theta_b - \theta_a}$. Afin de trouver le bon indice dans la table, il est possible de multiplier $\theta$ par ce coefficient et de prendre la partie entière du résultat. Il est également possible d'utiliser la partie décimale pour obtenir $\frac{\theta - \theta_a}{\theta_b - \theta_a}$. Ainsi le calcul du couple provenant du frein magnétique devient :

$$c = \theta \cdot \frac{1}{\theta_b - \theta_a} \tag{2.23}$$

$$k = \lfloor c \rfloor \tag{2.24}$$

$$l = frac(c) \tag{2.25}$$

$$\tau_{mbs}(\theta) = \tau_k + l \cdot (\tau_{k+1} - \tau_k) \tag{2.26}$$

### 2.5.3 Comparaison avec la référence

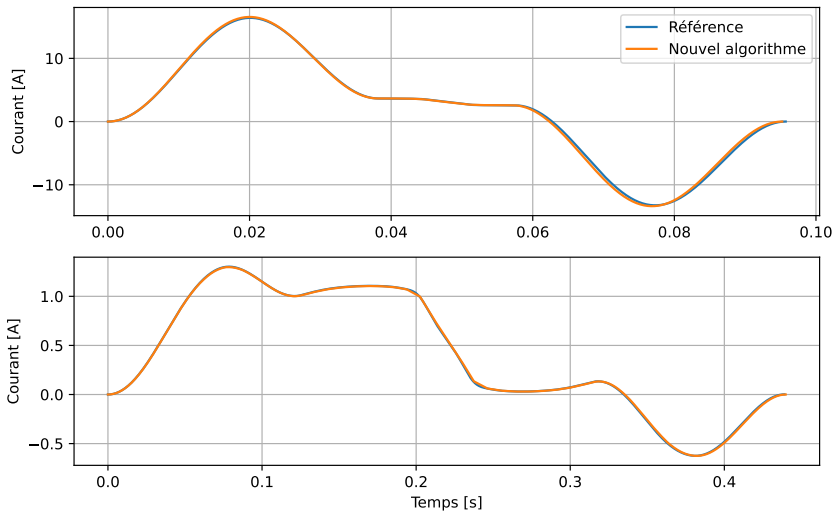


**Figure 2.7** – *Comparaison du courant généré avec l'algorithme de référence et la nouvelle version, avec* $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$ *en haut et* $\omega_{target} = 10\,\mathrm{rad\,s^{-1}}$ *en bas*

Les figures 2.7 et 2.8 présentent la comparaison de respectivement le courant et les trajectoires générées par l'algorithme actuel et la nouvelle version pour deux vitesses

constantes différentes. Les courbes sont très similaires. La différence provient principalement du fait que la version de référence n'atteint pas exactement la vitesse cible comme expliqué à la section 2.2.1. D'autres figures avec des paramètres différents sont disponibles à l'annexe C. La nouvelle version de l'algorithme semble donc valide.

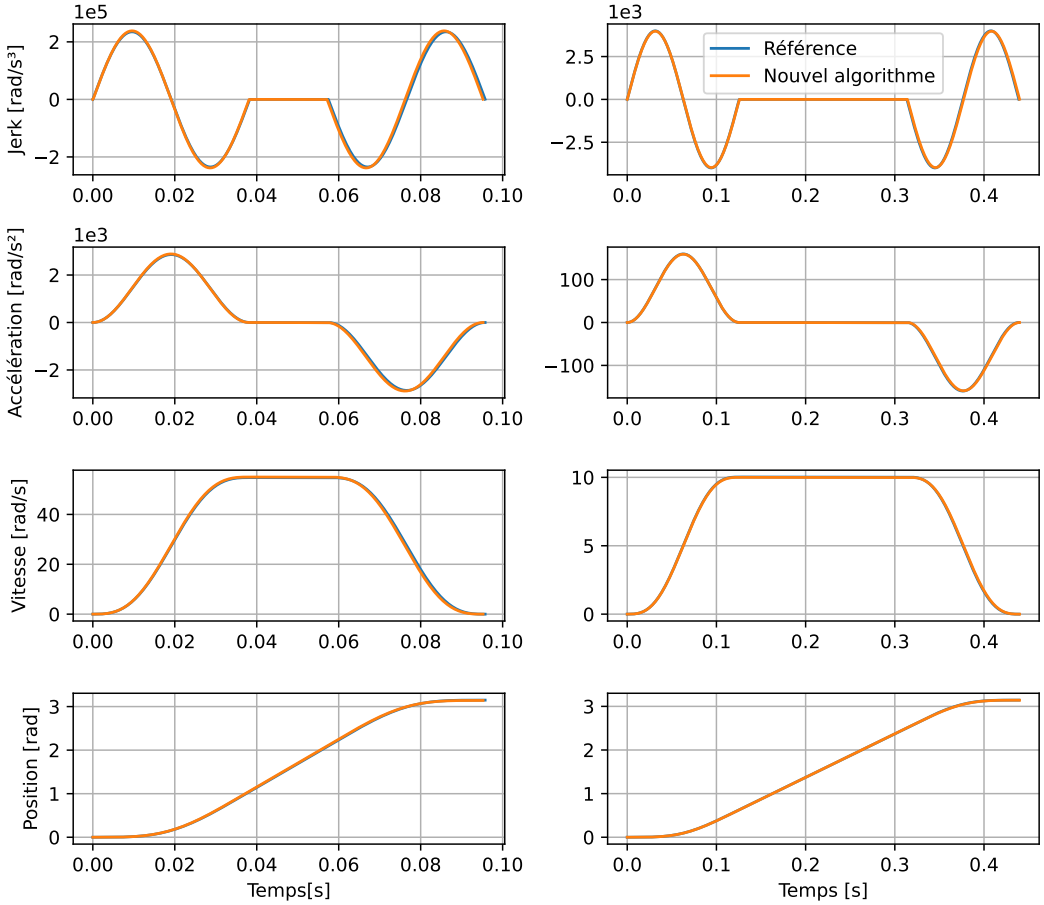


**Figure 2.8** – *Comparaison de la trajectoire générée avec l'algorithme de référence et la nouvelle version, avec* $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$ *à gauche et* $\omega_{target} = 10\,\mathrm{rad\,s^{-1}}$ *à droite*

### *2.5.4 Choix du type des variables*

Les calculs devant s'effectuer avec des nombres à virgules, le type des données utilisées est important. Deux options sont possibles : les nombres à virgule flottante ou les nombres à virgule fixe. Les calculs en nombres à virgule fixe utilisent moins de ressource, car il fonctionne de la même manière que les nombres entiers sans opération supplémentaire. Les nombres à virgule flottante permettent d'avoir une précision adaptative, ainsi qu'une plage de valeur plus élevée sans modifier la taille des données.

La table 2.1 présente les valeurs maximales prévues pour les paramètres d'entrée du système. Ces valeurs concernent les moteurs rotatifs et linéaires.

**Table 2.1** – *Valeurs maximales des paramètres d'entrée*

| Paramètre | Valeur maximale | |
|---|---|---|
| $\omega_{target}$ | 1000 | $\mathrm{rad\,s^{-1}}$ |
| $\theta_{acc}$ | 100 | rad |
| $\theta_{dec}$ | 100 | rad |
| $\theta_{total}$ | 100 | rad |
| $J_{tot}$ | 0.1 | $\mathrm{kg\,m^2}$ |
| $\mu_{dyn}$ | 0.5 | $\mathrm{N\,m\,s}$ |
| $\tau_{sta}$ | 1 | $\mathrm{N\,m}$ |
| $K_\tau$ | 2 | $\mathrm{N\,m\,A^{-1}}$ |

### *Niveaux de précision*

Les valeurs finales des trajectoires générées par l'algorithme sont des nombres à virgule fixe signés de 32 bits avec respectivement 16 bits pour les parties entière et fractionnaire. Les plages de valeurs utilisées étant connues à l'avance, il est possible de définir différents niveaux de précision en fonction de ce qui est calculé. La table 2.2 montre les différents niveaux de précision utilisés pour les calculs avec le nombre de bits pour la partie entière, le bit de signe compris basé sur les valeurs maximales données dans la table 2.1. Il a été arbitrairement décidé de garder le nombre de bits total égale pour tous les niveaux de précision, celui-ci sera discuté par la suite. Ces différents niveaux de précision sont utilisés pour les variables intermédiaires des calculs ainsi que pour les paramètres d'entrée.

**Table 2.2** – *Niveaux de précision utilisés pour les calculs en virgule fixe*

| Niveau | Partie entière | Plage de valeur |
|---|---|---|
| Normal | 16 | -32768 à 32767 |
| Moyen | 10 | -512 à 511 |
| Petit | 4 | -8 à 7 |
| Distance | 8 | -128 à 127 |
| Frein | 4 | -8 à 7 |
| Sinus | 4 | -8 à 7 |

La table 2.3 présente la correspondance entre les paramètres d'entrée et le niveau de précision utilisé. Le niveau *Sinus* est utilisé pour les valeurs d'entrée de la fonction sinus permettant de générer les étapes d'accélération et de décélération. Le niveau *Moyen* est utilisé pour certaines valeurs intermédiaires. Les variables pour l'accélération et le courant utilisent le niveau *Normal*.

**Table 2.3** – *Correspondance entre les paramètres d'entrée et le niveau de précision utilisé*

| Valeur | Niveaux de précision |
|---|---|
| $\omega_{target}$ | Normal |
| $\theta_{acc}$, $\theta_{dec}$ et $\theta_{total}$ | Distance |
| $J_{tot}$, $\mu_{dyn}$, $\tau_{sta}$ et $K_\tau$ | Petit |
| $\tau_{mbs}(\theta)$ | Frein |

### *Comparaison des types de données*

Afin de comparer la précision des calculs avec les nombres à virgule flottante et à virgule fixe, le paquet Python fpbinary [2] a été utilisé pour simuler les calculs avec des nombres à virgule fixe.

Les valeurs finales devant être retournées par la génération étant des nombres à virgule fixe sur 32 bits (16 bits entiers + 16 bits fractionnaires), les comparaisons sont effectuées avec ce type de donnée. Ceci permet de voir l'impact réel sur le système.

La figure 2.9 montre la différence entre les deux types de données pour la trajectoire générée avec $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$ et avec 32 bits pour les nombres à virgule fixe. Les deux versions obtenues sont relativement proches avec environ de 0.01 % de différence sur le courant. La différence est calculée en soustrayant les valeurs obtenues avec les nombres à virgule flottante à celles obtenues avec les nombres à virgule fixe. Ainsi, la forme de la différence de vitesse montre que la version à virgule fixe est légèrement plus rapide lors de la phase d'accélération.

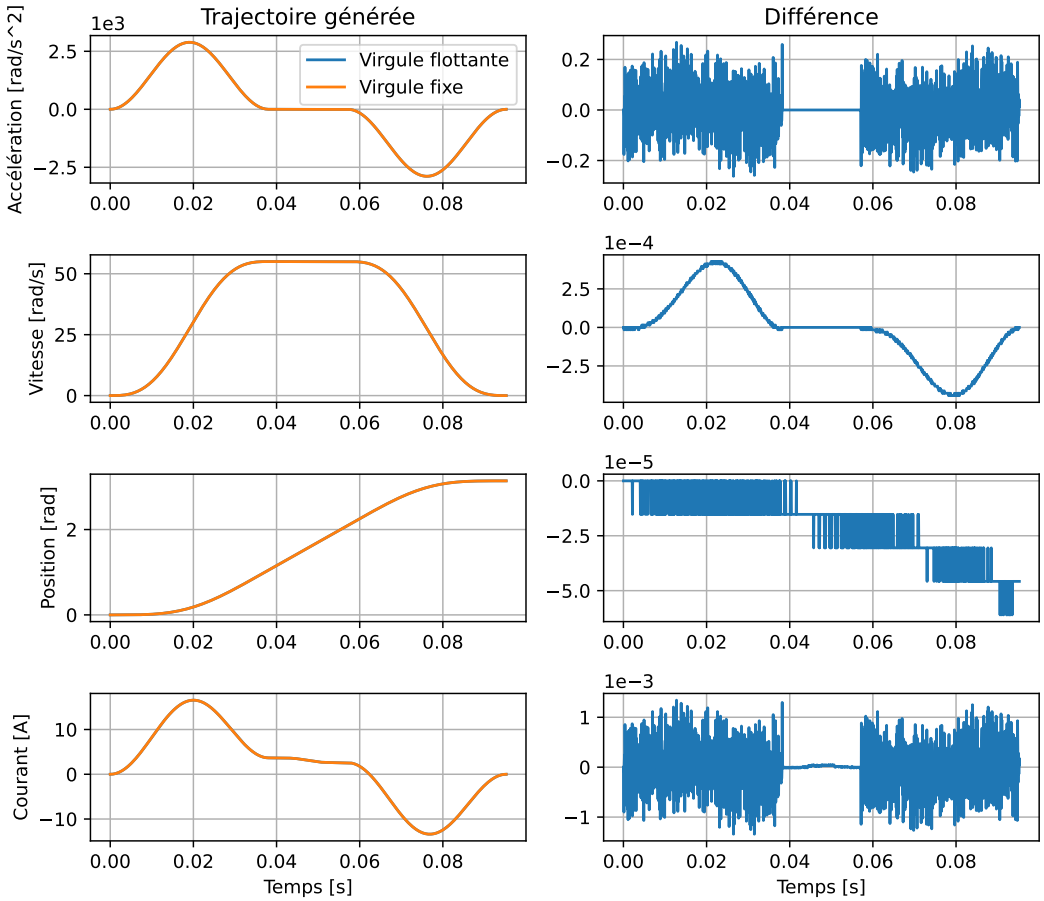


**Figure 2.9** – *Comparaison des trajectoires générées avec des nombres à virgule flottante et fixe sur 32 bits, avec* $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$

La figure 2.10 montre la même comparaison, mais avec 36 bits pour les nombres à virgules fixes. Les trajectoires sont maintenant quasiment égales. La différence

2. https://github.com/smlgit/fpbinary/tree/master

absolue maximale sur la vitesse et la position est d'environ $1.5 \cdot 10^{-5}$, ce qui correspond à la précision des 16 bits fractionnaires utilisés pour les comparaisons qui vaut $2^{-16} = 1.5259 \cdot 10^{-5}$. L'accélération est moins précise, ce qui s'explique par le fait qu'elle soit dérivée de la vitesse. Le courant n'est cependant pas trop affecté par cette différence et reste proche de la version à virgule flottante.

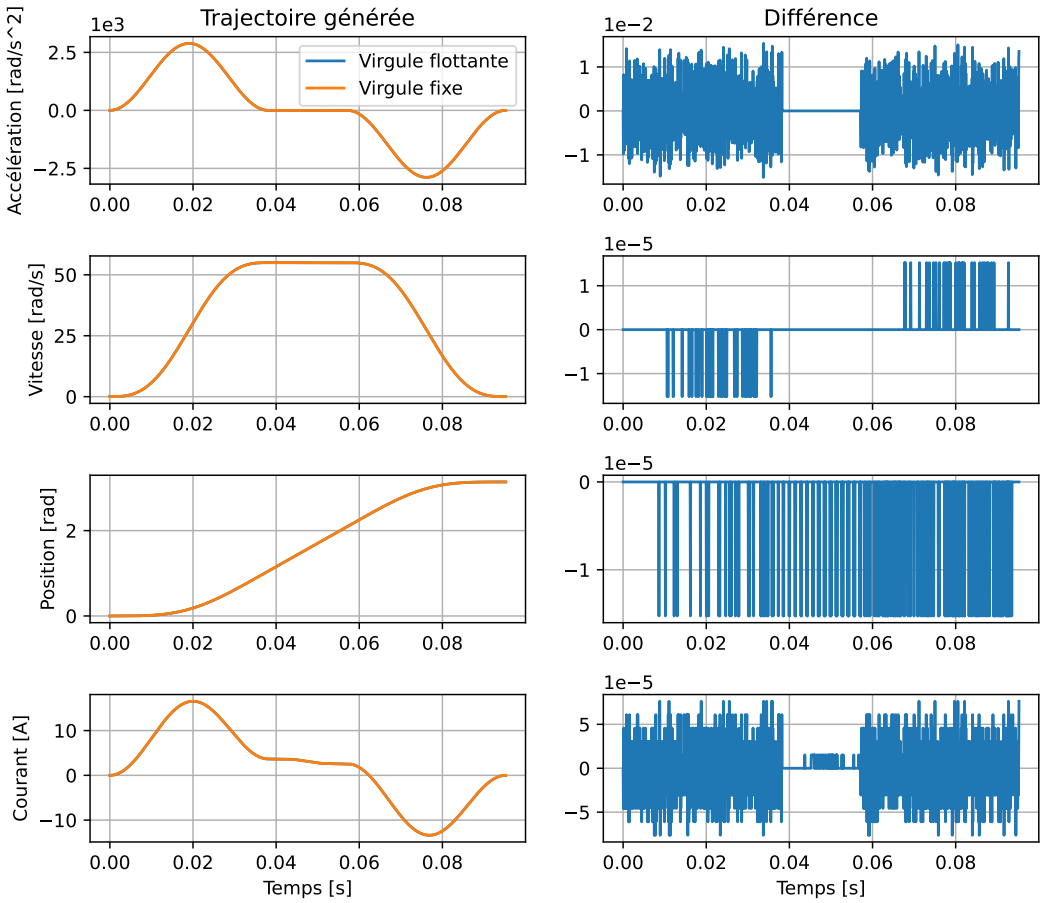


**Figure 2.10** – *Comparaison des trajectoires générées avec des nombres à virgule flottante et fixe sur 36 bits, avec* $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$

Les nombres à virgule fixe offrent donc une excellente précision avec une taille de 36 bits. C'est donc ce type de donnée qui sera utilisé lors de l'implémentation.

# Chapitre 3

# Conception

Ce chapitre présente les schémas des différents blocs du système de génération de trajectoire. Ces schémas ne sont pas exhaustifs de l'implémentation finale, mais permettent de comprendre le fonctionnement global et comment les signaux sont utilisés.

La génération se fait en amont de son utilisation, c'est-à-dire que la trajectoire est calculée avant que le moteur soit en mouvement. La durée des calculs n'est donc pas critique. Ainsi, les blocs de calculs fonctionnent majoritairement de manière séquentielle et une seule valeur est calculée à la fois. La mémoire RAM est utilisée pour enregistrer les valeurs calculées et un bloc est dédié à la lecture de ces valeurs lorsqu'un mouvement est en cours.

## *3.1 Système global*

La figure 3.1 présente le schéma global du système de génération de trajectoire. Les blocs *Trajectory calculator in/out* sont ceux qui vont calculer la trajectoire complète à effectuer pour respectivement l'aller et le retour puis les stocker dans la mémoire vive. Le bloc *Trajectory lookup* va lire la trajectoire dans la RAM, en fonction du sens, lorsque le moteur est en fonctionnement. Ces deux blocs sont détaillés dans les sections suivantes. Le bloc *Parameters inverter* va inverser les paramètres pour la génération *out* de la trajectoire comme décrit dans la section 2.5.1.

Les signaux d'entrée des blocs *Trajectory calculator in/out* sont tous les paramètres de la trajectoire présentés dans le chapitre 2.1 ainsi qu'un signal de contrôle pour démarrer le calcul. La LUT du frein magnétique contenant plusieurs valeurs, un signal de sortie permet d'indiquer l'index voulu dans le signal d'entrée. Les deux blocs de génération pouvant accéder à des adresses différentes de la LUT, les signaux sont doublés.

La sortie `trajectory_valid` est utilisée pour indiquer si la génération est valide et terminée, tandis que `parameters_invalid` indique si les paramètres sont invalides et que les limites physiques sont dépassées. La génération est considérée comme valide uniquement quand les deux blocs l'indiquent, c'est pour cela qu'une porte `AND` est utilisée pour le signal de sortie `trajectory_valid`. À l'inverse, les paramètres sont considérés invalides si au moins un des blocs l'indique, une porte `OR` est donc utilisée pour le signal de sortie `parameters_invalid`.

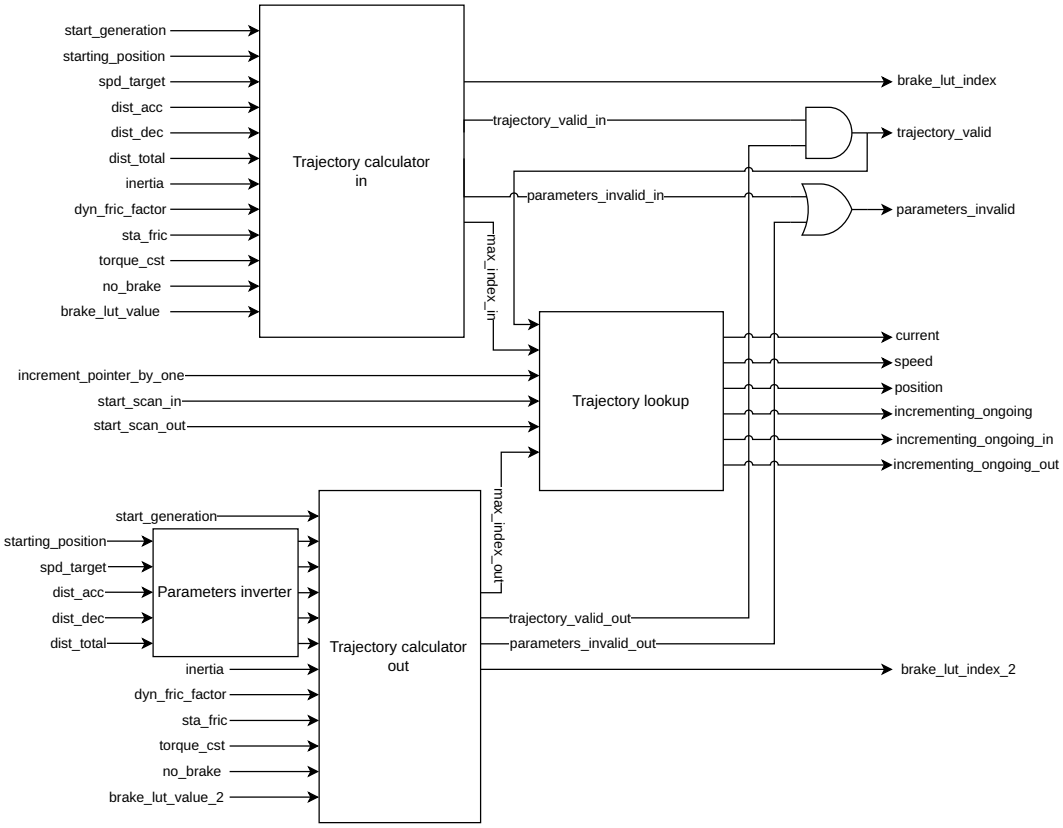


**Figure 3.1** – *Schéma global du système de génération de trajectoire*

Les signaux d'entrée et de sortie du bloc *Trajectory lookup* sont déjà utilisés dans le système actuel et permettent de contrôler le début d'un mouvement et les moments où la prochaine valeur doit être lue, en fonction de la PWM. Les signaux `max_index_in/out` sont utilisés pour connaître le nombre de valeurs dans la trajectoire.

## *3.2 Calcul de la trajectoire*

La figure 3.2 présente le schéma du bloc de calcul de la trajectoire. Trois blocs principaux sont présents :

1. *Pre-computing* : Pré-calculs les paramètres de la trajectoire
2. *Trajectory generator* : Génère la trajectoire (accélérations, vitesses, positions)
3. *Current converter* : Convertit la trajectoire en courant pour le moteur

Le signal `pre_comp_done` indique que tous les paramètres pré-calculés sont disponibles et que le calcul de la trajectoire peut commencer. Lorsque ce signal est inactif, les blocs *Trajectory generator* et *Current converter* doivent se réinitialiser pour la prochaine génération. Par défaut, le signal `pre_comp_done` est inactif et devient actif uniquement après qu'une impulsion du signal `start_generation` ait été détectée et que tous les paramètres aient été pré-calculés. Dès que le signal `start_generation` est activé, `pre_comp_done` doit être inactif pendant au moins 1 cycle d'horloge pour que les blocs se réinitialisent.

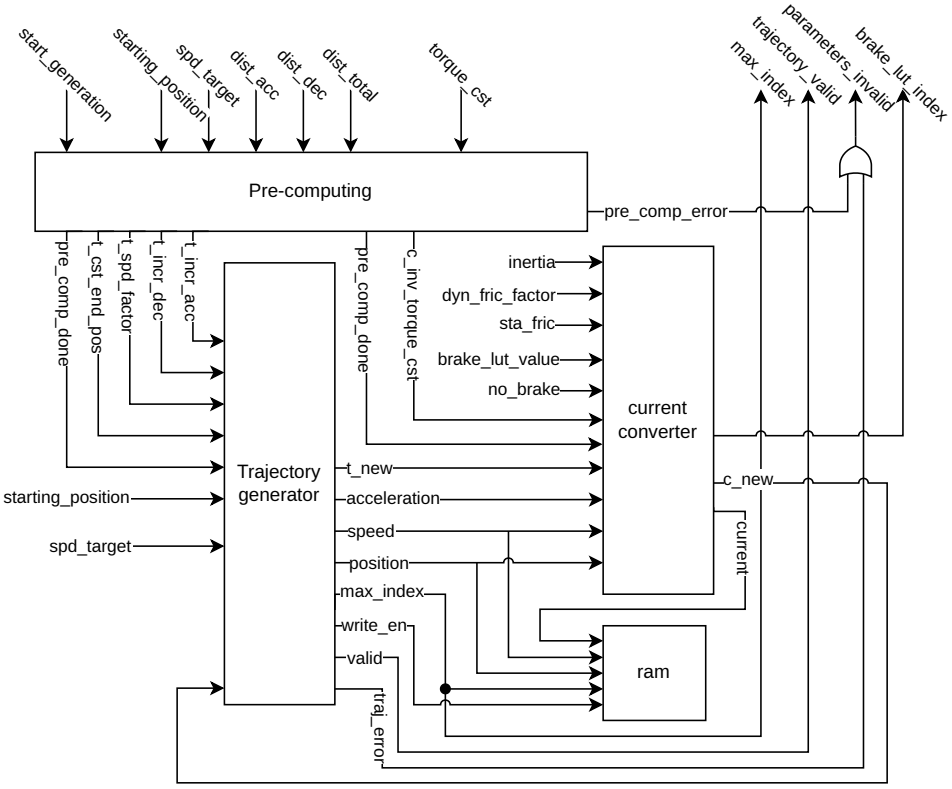


**Figure 3.2** – *Schéma du système de calcul de la trajectoire*

Le signal `t_new` indique que les prochaines valeurs d'accélération, vitesse et position de la trajectoire sont disponibles pour le bloc *Current converter*. Celui-ci doit alors convertir ces valeurs en courant et activer le signal `c_new`. Une fois ce signal actif, le bloc `trajectory_generator` peut activer le signal `write_en` pendant un cycle d'horloge pour écrire les données dans la mémoire et passer au calcul suivant. La valeur de `max_index` est utilisée pour indiquer à quel index de la RAM les données doivent être écrites.

Une fois que la trajectoire a été complètement générée le signal `trajectory_valid` est actif. Les signaux `pre_comp_error` et `traj_error` indiquent respectivement si une erreur est détectée lors des phases de pré-calcul et de calcul de la trajectoire. Lorsqu'une erreur est détectée, le signal `parameters_invalid` est activé.

### *3.2.1 Pre-computing*

La figure 3.3 présente le schéma du bloc *Pre-computing*. Les calculs effectués par ce bloc permettent de pré-calculer certaines valeurs qui sont utilisées à multiples reprises lors de la génération de la trajectoire.

Le bloc *Parameters validation* permet d'assurer que les paramètres d'entrée permettent une génération de trajectoire sans risque de blocage. Ainsi, une erreur est détectée si l'un des paramètres et égale à zéro ou si la vitesse et les différentes distances ne sont pas toutes positives ou toutes négatives. La vérification des dépassements de limites ce fait lors de la génération de trajectoire.

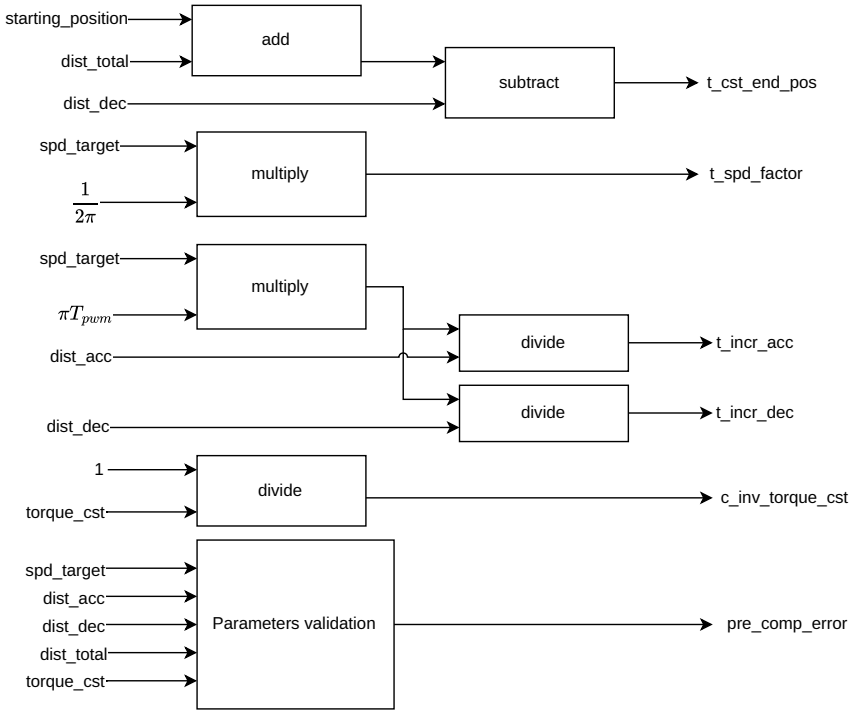


**Figure 3.3** – *Schéma du bloc* Pre-computing

### *3.2.2 Trajectory generator*

La figure 3.4 présente le schéma du bloc *Trajectory generator*. Les blocs *integrate* et *derivate* calculent respectivement la position et l'accélération en fonction de la vitesse en suivant les équations 2.16 et 2.17. Les valeurs précédentes nécessaires dans ces équations seront enregistrées dans des registres. Le signal `new_val` permet de déclencher le calcul de la prochaine valeur pour ces deux blocs et le résultat doit être valide au cycle d'horloge suivant.

Le bloc *changing speed* effectue l'équation 2.18 et 2.19 pour calculer la vitesse lors des étapes d'accélération et de décélération. Le signal `start_new` indique qu'une nouvelle étape commence dans la direction (accélération ou freinage) donnée. Le signal `next` indique que la prochaine valeur peut être calculée et `valid` que celle-ci a fini de l'être. `all_done` est actif quand toutes les valeurs de l'étape ont été calculées. La fonction sinus est calculée à l'aide du bloc *cordic*. Ce bloc sera une *Intellectual Property Core* (IP Core) fournie par Intel.

Le bloc *speed controller* sélectionne la vitesse à utiliser pour la prochaine valeur en fonction de l'étape actuelle et change d'étape quand la condition est atteinte. Il s'occupe également de synchroniser les différents blocs pour qu'une nouvelle vitesse ne soit pas envoyée avant que la position, l'accélération et le courant n'aient été calculés et sauvegardés en RAM. Le signal `max_index` est incrémenté uniquement quand les valeurs ont été enregistrée et est utilisé comme index d'écriture pour la mémoire. Il s'occupe également de vérifier que les limites n'aient pas été dépassées. Si c'est le cas, le signal `traj_error` est activé et le calcul de la trajectoire est arrêté.

### *3.2.3 Current converter*

La figure 3.5 présente le schéma du bloc *Current converter*. Les signaux de synchronisation avec le bloc *Trajectory generator* sont omis. Une nouvelle valeur de courant

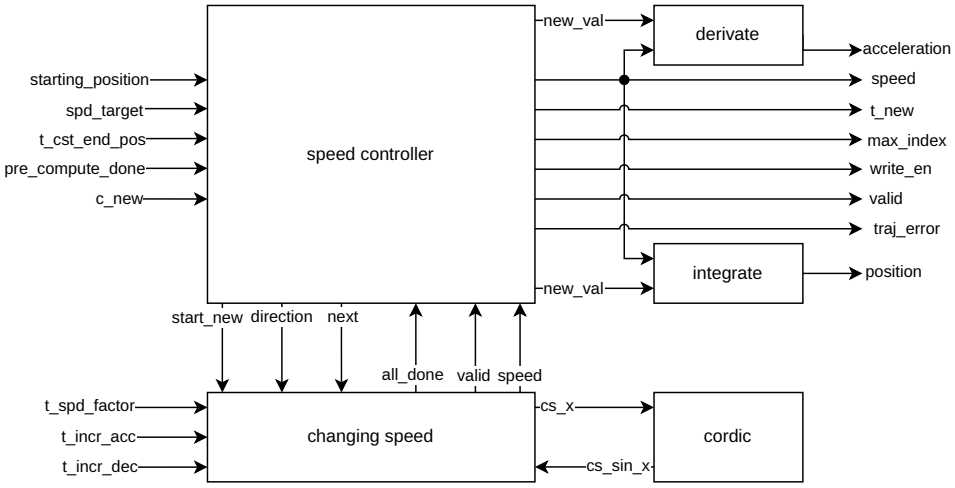


**Figure 3.4** – *Schéma du bloc* Trajectory generator

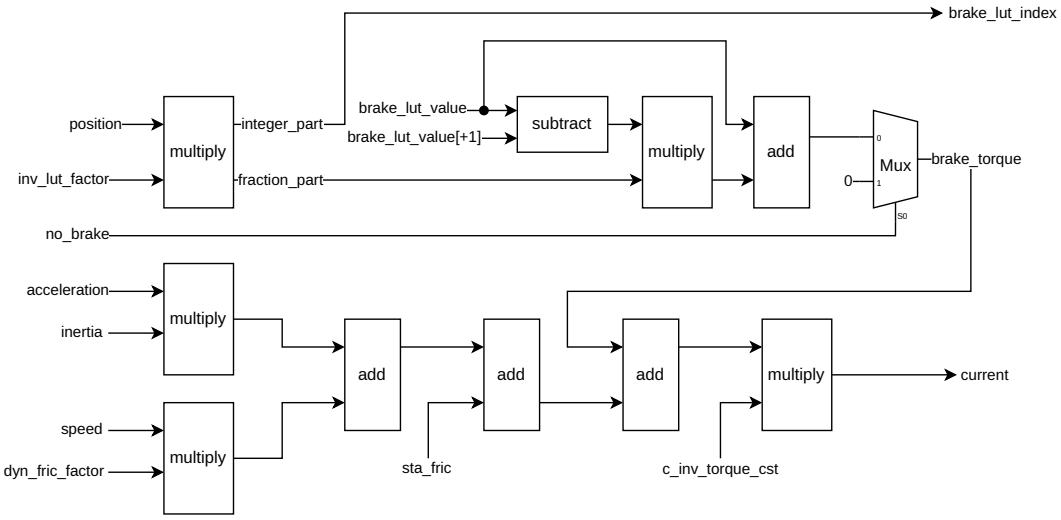


**Figure 3.5** – *Schéma du bloc* Current converter

est calculée dès que le signal `t_new` est actif. Quand tous les calculs sont effectués, le signal `c_new` doit être activé.

Ayant besoin de deux valeurs provenant de la LUT du frein magnétique, le signal de `brake_lut_index` est incrémenté après avoir lu la première valeur. La première valeur correspond à `brake_lut_value` et la seconde à `brake_lut_value[+1]`. Une synchronisation doit donc être ajoutée pour lire correctement les valeurs provenant de la LUT. L'index est corrigé pour ne pas dépasser la taille de la LUT en rebouclant sur la première valeur.

## *3.3 Relecture des trajectoires*

La figure 3.6 présente le schéma du bloc *Trajectory lookup*. Le bloc *Controller* contrôle le début, la direction et la fin du mouvement. En fonction de la direction, le bloc *Value select* retourne les bonnes valeurs provenant de la mémoire à l'index donné par le compteur. Ce dernier est incrémenté à chaque cycle de la PWM à l'aide du signal `increment_pointer_by_one` et est remis à zéro au début de chaque

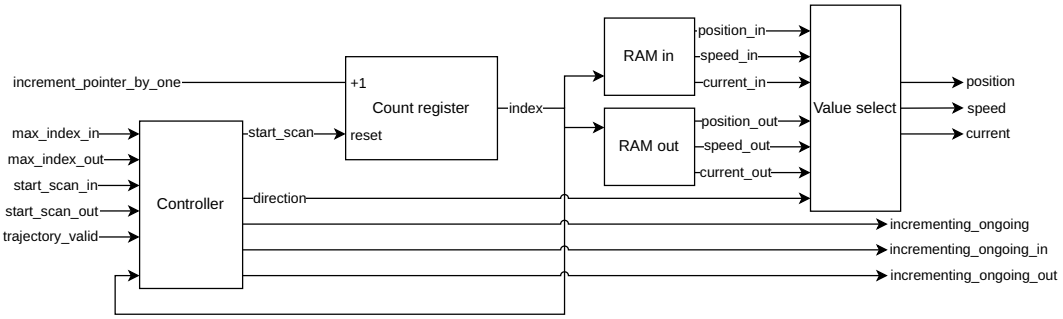


**Figure 3.6** – *Schéma du bloc* Trajectory lookup

nouveau mouvement.

Les signaux `incrementing_ongoing` indique si un mouvement est en cours en fonction la direction de celui-ci.

# Chapitre 4

# Implémentation

Ce chapitre présente les choix qui ont été fait lors de l'écriture des descriptions VHDL du projet. Trois entités ont été écrites, une principale qui est le point d'entrée du générateur et fait le lien entre les deux autres, une qui effectue le calcul de la trajectoire et une qui gère la relecture lors d'un mouvement.

Le paquet `ieee.fixed_pkg` est utilisé pour les nombres à virgule fixe avec le type `sfixed`. Celui-ci avait dû être modifié pour le rendre compatible lors de la synthèse et doit donc être compilé avant d'être utilisé, la compilation se fait dans la librairie `ieee_proposed`.

Afin de respecter les conventions du reste du projet, un *reset* synchrone est utilisé.

Pour de rendre la description la plus générique possible, une série de paramètres ont été définis. Ceux-ci sont utilisés pour définir les tailles des données, les limites physiques du système ainsi que d'autres paramètres. La table 4.1 présente les différents paramètres génériques. Les tailles des parties fractionnaires sont déduites des autres paramètres. Les limites d'accélération, vitesse et de courant prennent des valeurs positives, mais sont utilisées comme limite absolue. Les limites de positions sont définies légèrement plus loin que les valeurs possibles dans les trajectoires afin de permettre un petit dépassement dû à la précision des calculs. Les valeurs des paramètres sont sujets à changement lors de l'intégration dans le projet complet.

Tous les fichiers sources sont disponibles dans le dossier :

`dev/01_trajectory_generation/src_vhdl/`

La librairie `ieee_proposed` est disponible dans le dossier :

`dev/01_trajectory_generation/ieee_proposed/`

## *4.1 Entité principale*

L'entité principale est responsable de déclarer les sous-blocs pour le calcul de la trajectoire et la gestion du mouvement (relecture) et de faire les liens entre ces blocs et les entrées/sorties, selon le schéma 3.1. Il doit également changer le type des signaux d'entrée pour ceux à virgule fixe correspondant et inverser les paramètres pour la génération inverse de la trajectoire.

**Table 4.1** – *Paramètres génériques de la description VHDL*

| Nom | Description | Valeur |
|---|---|---|
| F_PWM | Fréquence de la PWM [Hz] | 16129 |
| MAX_NB_TRAJ_SAMPLE | Nbre maximum d'échantillons par trajectoire | 8192 |
| ADDRESS_WIDTH | Taille des adresses des échantillons | 13 |
| IO_DATA_SIZE | Nombre de bits des entrées/sorties du système | 32 |
| INTERN_DATA_SIZE | Nombre de bits des variables pour les calculs | 36 |
| NORMAL_INT_SIZE | Taille de la partie entière : précision *Normal* | 16 |
| MEDIUM_INT_SIZE | Taille de la partie entière : précision *Moyen* | 10 |
| SMALL_INT_SIZE | Taille de la partie entière : précision *Petit* | 4 |
| DIST_INT_SIZE | Taille de la partie entière : précision *Distance* | 8 |
| BRAKE_INT_SIZE | Taille de la partie entière : précision *Frein* | 4 |
| BRAKE_LUT_SIZE | Taille de la LUT du frein magnétique | 73 |
| BRAKE_ADDRESS_WIDTH | Taille de l'adressage de la LUT | 9 |
| BRAKE_LUT_MAX | Angle du dernier élément de la LUT [rad] | $2\pi$ |
| BRAKE_LUT_DELAY | Délai pour l'accès aux valeurs de la LUT | 3 |
| MAX_ACCELERATION | Limite maximale de l'accélération [$\mathrm{rad\,s^{-2}}$] | 20000 |
| MAX_SPEED | Limite maximale de la vitesse [$\mathrm{rad\,s^{-1}}$] | 1000 |
| MAX_POSITION | Limite maximale de la position [rad] | $2.1\pi$ |
| MIN_POSITION | Limite minimale de la position [rad] | $-0.1\pi$ |
| MAX_CURRENT | Limite maximale du courant [A] | 200 |

**Table 4.2** – *Précision utilisée pour les signaux d'entrées et de sorties*

| Signal | Précision |
|---|---|
| starting_position_i | Distance |
| spd_target_i | Normal |
| dist_acc_i | Distance |
| dist_dec_i | Distance |
| dist_total_i | Distance |
| inertia_i | Petit |
| dyn_fric_factor_i | Petit |
| sta_fric_i | Petit |
| torque_cst_i | Petit |
| brake_lut_value_i | Frein |
| brake_lut_value_2_i | Frein |
| feedforward_mp_cur_o | Normal |
| feedforward_mp_spd_o | Normal |
| feedforward_mp_pos_o | Normal |

Les signaux d'entrées et de sorties sont déclarés dans le listing 4.1. Tous ceux contenant des nombres sont de type `std_logic_vector` afin de garantir la compatibilité sans le paquet `fixed_pkg`. Ces nombres doivent respecter les niveaux de précisions (voir table 2.2) décrits dans la table 4.2. Les signaux correspondent aux entrées et sorties du schéma 3.1. Ceux commençant par `feedforward_mp_` sont les sorties du bloc *Trajectory lookup*. Ces noms permettent de garder la compatibilité avec ceux actuellement utilisés dans le projet. La liste des paramètres génériques de l'entité

est celle présentée dans la table 4.1.

```
port (
    clock_i             : in  std_logic;
    reset_i             : in  std_logic;
    start_generation_i  : in  std_logic;
    starting_position_i : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    spd_target_i        : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    dist_acc_i          : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    dist_dec_i          : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    dist_total_i        : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    inertia_i           : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    dyn_fric_factor_i   : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    sta_fric_i          : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    torque_cst_i        : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    no_brake_i          : in  std_logic;
    brake_lut_index_o   : out std_logic_vector(BRAKE_ADDRESS_WIDTH - 1 downto 0);
    brake_lut_index_2_o : out std_logic_vector(BRAKE_ADDRESS_WIDTH - 1 downto 0);
    brake_lut_value_i   : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    brake_lut_value_2_i : in  std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    increment_pointer_by_one_i   : in  std_logic;
    start_scan_in_i              : in  std_logic;
    start_scan_out_i             : in  std_logic;
    trajectory_valid_o           : out std_logic;
    parameters_invalid_o         : out std_logic;
    feedforward_mp_ongoing_o     : out std_logic;
    feedforward_mp_ongoing_in_o  : out std_logic;
    feedforward_mp_ongoing_out_o : out std_logic;
    feedforward_mp_cur_o         : out std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    feedforward_mp_spd_o         : out std_logic_vector(IO_DATA_SIZE - 1 downto 0);
    feedforward_mp_pos_o         : out std_logic_vector(IO_DATA_SIZE - 1 downto 0)
);
```

**Listing 4.1** – *Déclaration des signaux d'entrées et de sorties de l'entité principale*

## 4.2 Calcul de la trajectoire

L'entité qui calcule la trajectoire est responsable d'effectuer la génération complète d'une trajectoire en fonction des paramètres, selon le schéma 3.2. Elle contient également la déclaration de la mémoire RAM pour la trajectoire.

Plusieurs constantes contenant les valeurs connues à la compilation sont déclarées en virgule fixe pour pouvoir être utilisé facilement par la suite, tel que la fréquence et la période de la PWM ou encore $\pi$.

Le composant `altsyncram`[1] fourni par Intel est utilisé pour la RAM. Celui-ci est actuellement utilisé sous forme de ROM avec les trajectoires pré-calculées avant la synthèse. Ce composant est configuré pour fonctionner en mode *Dual port*, un port est utilisé pour l'écriture et l'autre pour la relecture lors d'un mouvement. La taille des données correspond à celle des sorties, soit 32 bits et le nombre d'éléments est de 8192 selon le paramètre générique correspondant. Au total, trois instances de ce composant sont utilisées pour la vitesse, la position et le courant.

1. https://www.intel.com/content/www/us/en/programmable/quartushelp/current/index.htm#hdl/mega/mega_file_altsynch_ram.htm

### 4.2.1 Pré-calcul

Selon le schéma 3.3, plusieurs divisions doivent être effectuées lors du pré-calcul des paramètres. Pour cela, l'IP Core `lpm_divide` [2] est utilisée. Une division entre deux nombres à virgule fixe est normalement effectuée en ajoutant un certain nombre de bits à la partie fractionnaire du numérateur en fonction des tailles des deux opérandes et du résultat voulut. Cependant, dans le cas du calcul des incréments, cette opération nécessite plus de 64 bits ce qui dépasse la limite de l'IP Core. Pour contourner ce problème, les dénominateurs sont d'abord inversés puis multiplié par le numérateur. Une division est un processus long qui a un impact non négligeable sur la fréquence maximale atteignable par la FPGA. L'IP Core `lpm_divide` permet de définir un délai, en nombre de cycles d'horloge, entre le moment où les opérandes sont mises à jour et le moment où le résultat est disponible. Ce délai génère un processus utilisant un pipeline et permet ainsi d'augmenter la fréquence de fonctionnement de la FPGA. Celle-ci étant fixée à 40 MHz pour le projet, un délai de 15 cycles permet d'atteindre les performances voulues. Cette valeur peut être modifiée à l'aide de la constante `DIVIDE_DELAY`.

Un processus synchrone est créé pour effectuer les pré-calculs. Une machine de trois états est utilisée pour gérer les différentes étapes des calculs. La figure 4.1 présente les différents états ainsi que les calculs effectués lors des transitions. La validation des paramètres s'effectue lors de l'attente d'une nouvelle génération. Tant que ceux-ci ne sont pas valides, les pré-calculs ne peuvent pas être effectués et le signal `pre_comp_error` est actif. Le signal `pre_comp_done` est désactivé lorsqu'une nouvelle génération est demandée.

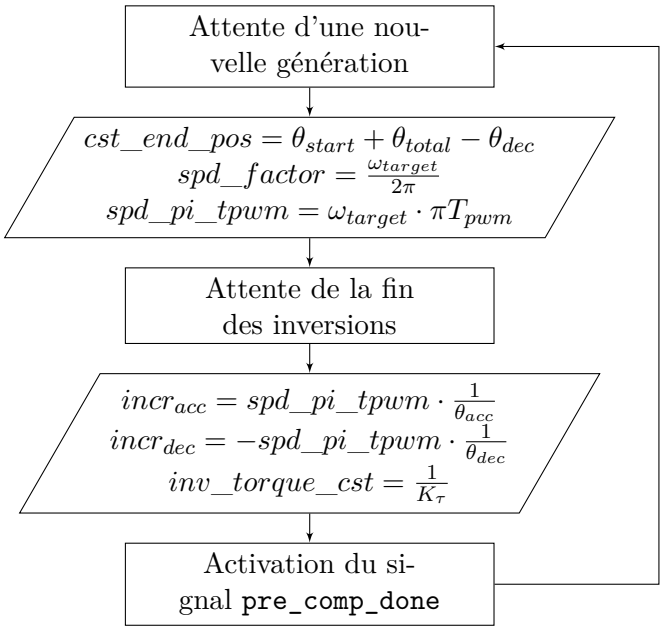


**Figure 4.1** – *Machine d'état du bloc* Pre-computing

### 4.2.2 Calcul des changements de vitesse

Lors des étapes d'accélération et de décélération de la trajectoire, les blocs *Changing speed* et *Cordic* du schéma 3.4 sont utilisés. Le premier est responsable d'effectuer

2. https://www.intel.com/content/www/us/en/docs/programmable/683490/20-3/lpm-divide-divider-intel-fpga-ip-core.html

les équations 2.18 et 2.19 pour générer la vitesse et le second est utilisé pour obtenir le sinus de l'équation 2.19.

Le bloc *Cordic* utilise l'IP Core `altera_cordic` [3] fourni par Intel, générée à l'aide de Quartus. Celle-ci est configurée pour calculer les sinus avec des nombres signés de 33 bits pour la partie fractionnaire en entrée et 32 en sortie, soit respectivement 36 bits et 34 bits au total. La plage des valeurs d'entrées est de $[-\pi, \pi]$. Une adaptation des calculs doit être faite, car l'équation 2.18, qui est utilisée pour le sinus, a une plage de $[0, 2\pi]$. Pour cela, un offset de $\pi$ est soustrait aux valeurs de départ $x_{acc}(0)$ et $x_{dec}(0)$ et l'équation 2.19 doit être modifiée par 4.1. La précision *Sinus* des nombres à virgules définie à la table 2.2 est également décalée de 1 bit vers la partie fractionnaire et contient donc 3 bits pour la partie entière.

$$\omega(k) = (x(k) + \pi + sin(x(k))) \cdot \frac{\omega_{target}}{2\pi} \tag{4.1}$$

Tout comme la division, pour obtenir la bonne fréquence de fonctionnement de la FPGA, un délai de 18 cycles d'horloge est ajouté au bloc *Cordic*. La constante `CORDIC_DELAY` permet de modifier cette valeur, il faut toutefois mettre à jour l'IP Core avec le nouveau délai.

Les fichiers générés par Quartus pour l'IP Core sont disponibles dans le dossier :

`dev/01_trajectory_generation/src_vhdl/traj_sincos/`

Le bloc *Changing speed* est implémenté à l'aide d'un processus synchrone. Celui-ci est décomposé en plusieurs états comme montrer dans la figure 4.2. En tout temps, si une nouvelle étape de changement de vitesse est demandée, le processus recommence la séquence, permettant ainsi de recommencer une génération si la précédente n'est pas terminée.

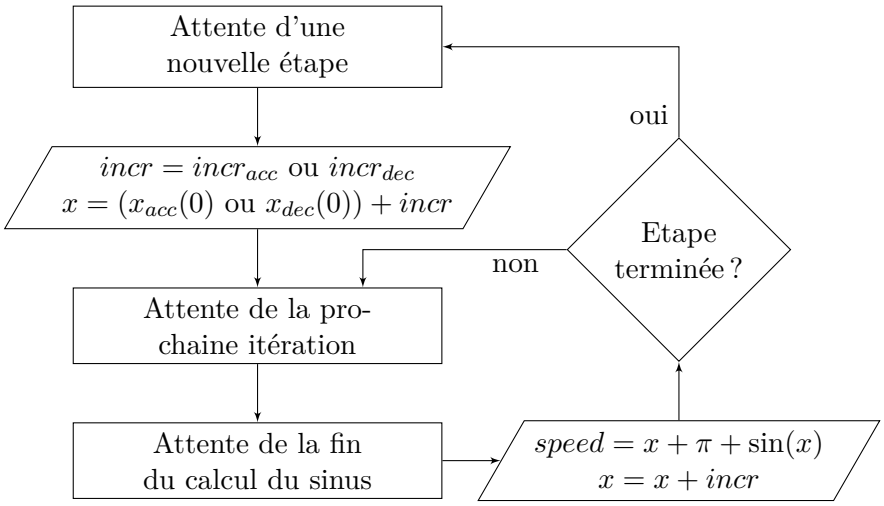


**Figure 4.2** – *Machine d'état du bloc* Changing speed

3. https://www.intel.com/content/www/us/en/docs/programmable/683808/current/altera-cordic-ip-core-user-guide.html

### 4.2.3 Générateur de la trajectoire

Un processus synchrone est utilisé pour générer la trajectoire. Celui-ci regroupe les blocs *Speed controller*, *Derivate* et *Integrate* du schéma 3.4. Il s'occupe donc de gérer les différentes étapes de la trajectoire en choisissant la vitesse à utiliser, soit via le bloc *Changing speed*, soit avec la vitesse constante. Il va également calculer les valeurs de position et d'accélération et indiquer au bloc *Current converter* qu'une nouvelle donnée est disponible. Pour cela, le processus est décomposé en plusieurs états comme montrer dans la figure 4.3. En tout temps, si le signal `pre_comp_done` est inactif, le processus est réinitialisé à la première étape, ce qui permet d'interrompre la génération de la trajectoire lorsqu'une nouvelle est demandée. Lorsque le calcul du courant est terminé, la vérification du respect des limites physiques est faite et si elles le sont, l'écriture dans la mémoire est activée, sinon la génération est interrompue et le signal d'erreur activé.

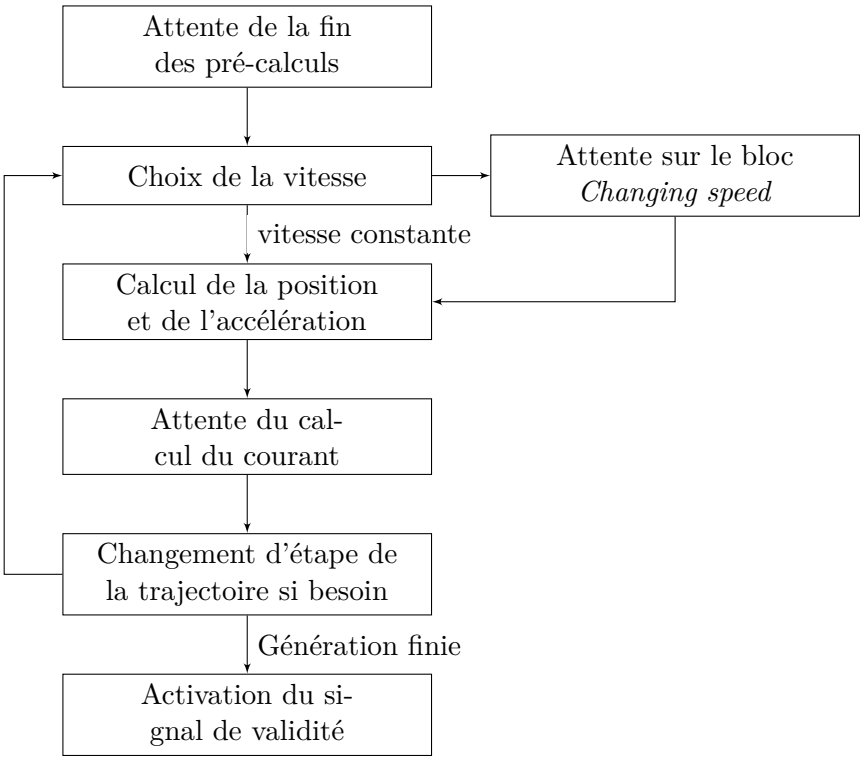


**Figure 4.3** – *Machine d'état du bloc* Trajectory generator

### 4.2.4 Conversion du courant

La conversion des valeurs de la trajectoire en courant est effectuée par le bloc *Current converter* présenté par le schéma 3.5. Celui-ci est implémenté à l'aide d'un processus synchrone divisé en plusieurs états comme le montre la figure 4.4. Le calcul de l'index correspond aux équations 2.23, et 2.24 2.25. Les deux attentes des valeurs du frein correspondent au délai entre la modification de l'index et la mise à jour de la valeur provenant de la LUT. Ce délai est défini par le paramètre générique `BRAKE_LUT_DELAY`. Afin de ne pas dépasser la taille de la table de valeur, l'index est corrigé si celui-ci dépasse la valeur maximale ou minimale en rebouclant sur la valeur opposée.

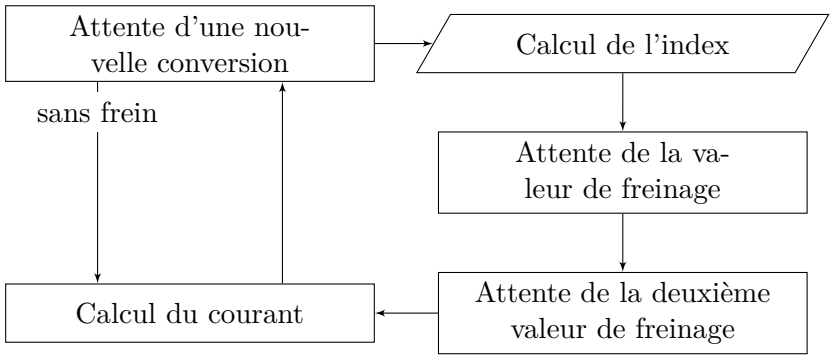


**Figure 4.4** – *Machine d'état du bloc* Current converter

## 4.3 Relecture des valeurs lors d'un mouvement

L'entité de relecture des données correspond au bloc *Trajectory lookup* présenté dans la section 3.3.

Un premier processus combinatoire choisit les valeurs en fonction de la direction du mouvement. Si la trajectoire n'est pas indiquée comme étant valide, les valeurs sont mises à zéro pour éviter tout problème.

Un deuxième processus combinatoire contrôle le départ d'un nouveau mouvement en activant les signaux `start_scan_s` et `motion_direction_s` en fonction du sens voulu.

Un troisième processus synchrone sauvegarde dans un registre la direction du mouvement actuel et active les signaux `incrementing_ongoing_*`. Ceux-ci sont désactivés lorsque l'index final est atteint.

Un dernier processus synchrone permet d'implémenter le compteur pour l'index. Ce compteur est remis à zéro lorsqu'un nouveau mouvement est demandé et incrémenté en fonction du signal `increment_pointer_by_one` si un mouvement est en cours.

## 4.4 Résultat de la synthèse

Afin de vérifier la validité des descriptions VHDL, une synthèse avec Quartus Prime est effectuée. Pour assurer que l'analyse des temps de propagation soit correcte, une entité supplémentaire est créée, celle-ci contient une instance de l'entité principale et les mêmes signaux d'entrée et sortie. Tous les signaux sont connectés à des registres, ce qui permet d'assurer que tous les chemins de propagation des signaux soient d'un registre à un autre et non un pin d'entrée ou de sortie.

Le fichier utilisé pour ajouter les registres est disponible :

`dev/01_.../srv_vhdl/trajectory_generation_top_register.vhd`

La synthèse a été faite pour une FPGA de la famille Arria V avec le modèle `5AGXMB1G4F40C4`. La table 4.3 présente les ressources utilisées de la FPGA après la synthèse. La dernière colonne indique le pourcentage d'utilisation totale actuelle du projet où le générateur devra être intégré.

Le nombre de bits mémoires correspond à la taille attendue qui est de $2 \cdot 3 \cdot 8192 \cdot 32 = 1'572'864$, la différence provient des divisions qui utilisent de la mémoire en interne. Les blocs *Digital Signal Processor* (DSP) sont principalement utilisés par les

**Table 4.3** – *Ressources utilisées après la synthèse*

| Type | Nombre | Pourcentage | Actuellement utilisé |
|---|---|---|---|
| Module logique (ALM) | 19'731 | 17% | 26% |
| Registres | 14'257 | - | - |
| Bit mémoire | 1'573'425 | 10% | 78% |
| Bloc DSP | 92 | 10% | 11% |

multiplications. La mémoire utilisée actuellement par le projet est principalement dû aux différentes trajectoires pré-calculées qui sont ajoutées au projet et qui pourront être supprimées lors de l'intégration du générateur.

Les ressources utilisées sont donc suffisamment basses pour permettre l'intégration du générateur dans le projet. La fréquence maximale de fonctionnement pour l'horloge du système est de 51.6 MHz ce qui est supérieur à la fréquence de 40 MHz utilisées. Le générateur peut donc être intégré dans le projet. Les signaux d'entrées et de sorties de l'entité principale n'étant pas forcément des sorties directes d'un registre, il est possible que lors de l'intégration la fréquence de fonctionnement soit plus petite auquel cas il faudra rajouter des registres sur les chemins de propagation posant problème.

La majorité des ressources utilisées proviennent des IP Core pour les divisions et *Cordic*. Les divisions utilisent en tout environ 9700 ALM et 7000 registres et les blocs *Cordic* environ 8000 ALM et 5600 registres. Ces blocs étant six fois pour les divisions et deux fois pour *Cordic*, il serait possible de réduire le nombre de ressources utilisées, si nécessaire, en regroupant leur utilisation et ajoutant des signaux de synchronisation.

# Chapitre 5

# Banc de test

Ce chapitre présente les bancs de test qui ont été développés pour tester les différentes descriptions VHDL. La librairie Python Cocotb [1] a été utilisée. Cette librairie est faite pour écrire des tests vérifiant le comportement de description écrite en VHDL et en SystemVerilog. Un simulateur, comme QuestaSim ou Riviera-Pro, est tout de même nécessaire, car le paquet doit si interfacer, cependant plusieurs simulateurs sont compatibles. Cependant, la compatibilité des bancs de test a pu être vérifiée uniquement pour QuestaSim. Les autres simulateurs ne sont pas garantis de fonctionner sans modification. Des scripts Python permettant de générer les trajectoires ayant été écrits lors de l'analyse des algorithmes, le choix de cette librairie a permis de limiter la réécriture afin d'avoir un modèle de comparaison.

De la même manière que des tests écrits en VHDL ou SystemVerilog, Cocotb fonctionne avec un système de tâches, appelées coroutines, qui s'exécutent de manière asynchrone. Chacune de ces coroutines peuvent alors lire ou stimuler les signaux du *Device Under Test* (DUT) entre chaque pas de simulation. Il est également possible d'attendre sur des événements, comme un flanc montant d'un signal. La classe `CoroutineBase` permet, via un héritage, d'implémenter d'autres classes ayant des fonctions `start` et `stop` qui démarre et arrête une coroutine. Les classes enfants doivent surcharger la fonction `_run` avec l'implémentation qui sera exécutée.

Trois bancs distincts ont été écrits permettant de vérifier le fonctionnement des blocs *Trajectory lookup* et *Trajectory calculator* indépendamment et un dernier pour l'entité principale qui regroupe ces deux blocs.

## *5.1 Modèle*

Un modèle du générateur de trajectoire a été développé en Python. Celui-ci permet d'avoir une référence qui sera comparée avec les sorties du DUT lors de la simulation. De la même manière que l'implémentation, le modèle est composé de deux classes principales. La première, `TrajectoryGenerator`, permet de générer une trajectoire et la deuxième, `TrajectoryLookup`, permet de lire la trajectoire comme le bloc *Trajectory lookup*. Une troisième classe, `TrajectoryDataDirection`, est utilisée pour stocker les données de la trajectoire.

L'algorithme utilisé par la classe `TrajectoryGenerator` pour générer la trajectoire est le même que celui utilisé dans le bloc *Trajectory calculator*. La différence majeure

1. https://www.cocotb.org

est que le modèle effectue les calculs en virgule flottante, permettant de ne pas se soucier de la précision. Lors de la génération, les données sont enregistrées dans une instance de la classe `TrajectoryDataDirection` et les deux directions sont générées en même temps. Lorsqu'une trajectoire est considérée comme invalide, dû aux limites physiques, la génération est tout de même effectuée, permettant ainsi de vérifier les données générées par le bloc *Trajectory calculator* tant que l'invalidité n'est pas détectée.

La classe `TrajectoryLookup` effectue la lecture de la trajectoire. Lorsqu'un mouvement est demandé, les données de la trajectoire sont lues dans l'ordre et en fonction de la direction puis ajoutée à une file d'attente.

La classe `TrajectoryDataDirection` contient les données des trajectoires pour chacune des directions. Pour cela, deux instances de la classe `TrajectoryData` sont utilisées. Cette dernière simule la mémoire contenant les trajectoires. Trois tableaux sont utilisés pour stocker les données de position, vitesse et courant, sous forme de nombre à virgule fixe de la même taille que celle utilisée pour les sorties du DUT permettant ainsi de comparer les données avec le même niveau de précision.

Les différents fichiers contenant ces classes sont disponibles dans le dossier :

`dev/01_trajectory_generation/model/`

## 5.2 Environnement de simulation

La figure 5.1 présente l'environnement de simulation utilisé. Le séquenceur est chargé d'indiquer quelles actions doivent être effectuées par le pilote. Celui-ci stimule les signaux du DUT et appel les fonctions nécessaires du modèle. Le DUT correspond à l'entité qui est testée. Le moniteur est chargé de lire les signaux du DUT et de les transmettre au scoreboard. Ce dernier compare les données reçues du moniteur avec le modèle pour s'assurer que le DUT fonctionne correctement. La transmission des données entre les différents éléments se fait par des files d'attente ou la classe `TrajectoryDataDirection`.

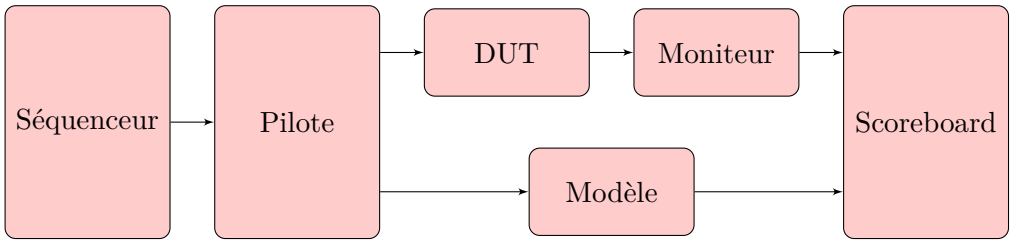


**Figure 5.1** – *Environnement de simulation*

Le séquenceur est implémenté dans la fonction principale du banc de test correspondant.

Plusieurs pilotes, moniteurs et scoreboards ont été développés afin de pouvoir être réutilisé par les différents tests, limitant la duplication de code. Chacun d'eux hérite de la classe `CoroutineBase` et son ainsi exécuter dans des coroutines séparées. Les signaux utilisés sont passés lors de l'initialisation de la classe via un dictionnaire, permettant ainsi de les rendre indépendant des signaux réels. Plus de détails sur l'implémentation de ces classes sont disponibles dans la section 5.3 présentant les tests en détails

La classe `SimulationEnvironment` permet de regrouper tous les pilotes, moniteurs et scoreboards utilisés pour la simulation. Elle contient une liste pour chaque type et des fonctions permettant d'y ajouter un élément. La fonction `run` lance la simulation, ce qui a pour effet de démarrer toutes les coroutines. Une horloge système est alors lancée et une procédure de remise à zéro de la simulation est exécutée. La période de l'horloge et la durée d'activation du signal `reset` sont définissables via les arguments à l'instanciation de la classe. La fonction `run` attend ensuite que la simulation soit terminée avant de d'arrêter toutes les coroutines.

Pour détecter que la simulation est terminée, un système de *timeout* est mis en place dans la classe `SimulationEnvironment`. Celui-ci vérifie à intervalle régulier qu'une action a été effectuée. Si aucune action n'a été effectuée durant l'intervalle actuel, la simulation est considérée comme terminée. Pour indiquer une action, la fonction `beat` doit être appelée. Celle-ci est alors utilisée par les drivers lorsqu'une séquence est en cours et par les moniteurs lorsque des données sont reçues. Cette méthode a aussi l'avantage de pouvoir détecter si la simulation est bloquée, par exemple si un driver attend une réponse du DUT qui ne viendra jamais. Il n'est cependant pas possible de distinguer entre une fin normale ou par blocage. Pour cela, les scoreboards implémentent une fonction supplémentaire, `end_simulation`, qui permet de faire des vérifications après avoir détecté la fin de la simulation. Par exemple, en cas de blocage, certaines files d'attentes peuvent ne pas être complètement vidées.

Les drivers, moniteurs, scoreboards et les deux classes présentées sont disponibles dans le dossier :

`dev/01_trajectory_generation/src_tb/environment/`

## 5.3 Test des entités

Chaque entité a un banc de test qui permet de vérifier son fonctionnement indépendamment des autres. Pour cela, plusieurs séquences de test ont été développées. Celles-ci regroupent plusieurs pilotes, moniteurs et scoreboards.

Pour lancer les tests, chaque banc a un Makefile permettant de lancer la compilation et la simulation. Ces Makefile compilent toutes les librairies nécessaires. Pour lancer les tests, il suffit d'exécuter la commande `make` depuis le dossier du banc de test. La commande `make GUI=1` permet de lancer la simulation avec l'interface graphique du simulateur permettant d'analyser les signaux. Il faudra alors lancer la simulation manuellement depuis l'interface, par exemple avec la commande `run -all`. Plus de détail sur l'utilisation de ces Makefile sont disponibles dans la documentation de Cocotb.

### 5.3.1 Valeurs d'entrée

Plusieurs groupes de paramètres d'entrée pour les trajectoires et les profils de courant ont été définis. Chacun de ces groupes fournit une trajectoire valide et à un nom distinct pour le différencier des autres.

Il y a douze groupes de paramètres pour les trajectoires. Ils ont été définis à partir de ceux utilisés dans les générations Matlab et ont été choisi pour couvrir un maximum de cas possible avec des vitesses basses et élevées et des distances d'accélération et de freinage différentes.

Les profils de courant sont définis par trois groupes de paramètres. Les valeurs ont également été inspirées par les générations Matlab et permettent de couvrir les cas où il n'y a pas de friction statique ou dynamique et différentes valeurs d'inerties.

Des combinaisons de ces groupes sont définies pour tester un maximum de cas sans allonger inutilement la durée des tests. Chaque groupe de paramètres des trajectoires est combiné avec deux profils de courant différents, une fois avec et une fois sans frein. Les profils de courant sont utilisés de manière égale dans ces différentes combinaisons.

En plus de ces combinaisons valides, des paramètres invalides sont définis pour tester le respect des limites physiques du système et la détection d'erreur. Dix groupes sont définis où la vitesse, les différentes valeurs de déplacement et les profils de courant sont mis à zéro où des valeurs trop élevées.

Ces valeurs sont définies dans le fichier :

`dev/01_trajectory_generation/model/trajectory_parameters.py`

### *5.3.2 Test de l'entité* Trajectory lookup

Le test de l'entité *Trajectory lookup* vérifie que les données de la trajectoire sont correctement lues et retransmises. Pour cela, une trajectoire est générée à l'aide du modèle, puis un mouvement du moteur est simulé. Ce bloc n'étant pas dépendant des paramètres d'entrée, toutes les trajectoires sont testées, mais seuls un profil de courant est utilisé et le frein n'est pas désactivé, afin de réduire le temps total de simulation. Un profil invalide est également testé afin de s'assurer que le bloc ne fait rien dans ce cas.

Les fichiers spécifiques à ce banc sont disponibles dans le dossier :

`dev/01_trajectory_generation/src_tb/lookup/`

#### *Pilote*

Deux pilotes, `TrajectoryRamDriver` et `LookupDriver`, sont utilisés pour ce test permettant de contrôler les différents signaux d'entrée du bloc.

Le premier simule la lecture depuis la mémoire contenant les trajectoires. Les valeurs de position, vitesse et courant sont mises à jour en fonction de l'index reçu. La classe `TrajectoryData` est utilisée pour stocker les valeurs générées par le modèle. Ce driver est instancié deux fois pour les deux directions du moteur.

Le pilote `LookupDriver` effectue les séquences permettant de tester la lecture de la trajectoire lors d'un mouvement. La figure 5.2 présente la séquence effectuée. L'attente de plusieurs cycles permet de simuler la période de la PWM. Lors du démarrage et avant chaque attente pour l'incrémentation, une action est notifiée à l'environnement de simulation via la fonction `beat`.

#### *Moniteur*

Le moniteur `LookupMonitor` est utilisé pour lire les données provenant de la lecture d'une trajectoire. Il attend qu'un nouveau mouvement du moteur soit démarrer pour ensuite lire les données de position, vitesse et courant après chaque incrémentation de l'index tant que le mouvement est en cours. Ces données sont mises dans une file d'attente qui est ensuite lue par le scoreboard. Les signaux indiquant si le mouvement est en cours sont également transmis. Un certain délai est attendu avant de lire les données après une incrémentation. Cela permet de laisser le temps au bloc de mettre à jour les données.

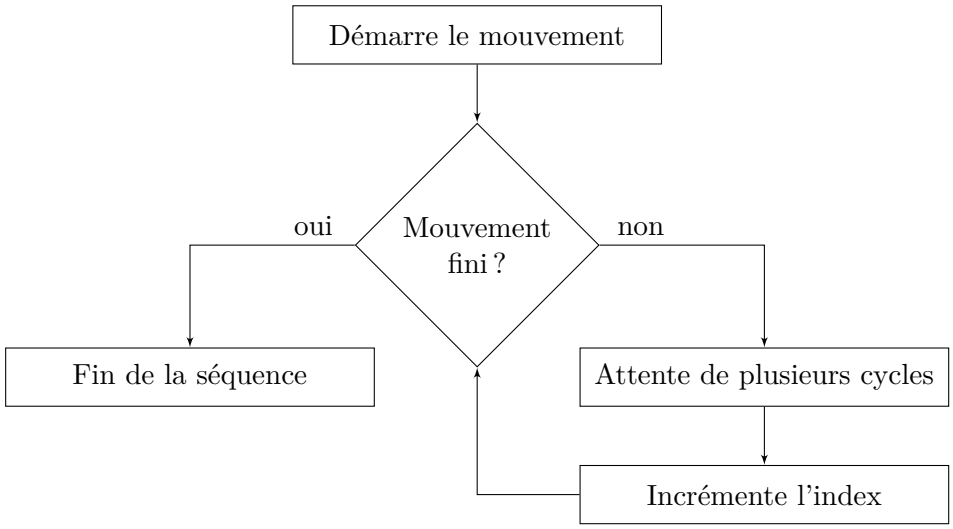


**Figure 5.2** – *Séquence du pilote* `LookupDriver`

***Scoreboard***

Le scoreboard `LookupScoreboard` est utilisé pour valider la trajectoire lue. Il compare les données reçues du moniteur avec celles générées par le modèle. Deux files d'attente sont utilisées, une provenant du moniteur et l'autre de la référence. Lorsque la simulation se termine, le scoreboard vérifie que les deux files d'attente soient vides.

### *5.3.3 Test de l'entité* Trajectory calculator

Le test de l'entité *Trajectory calculator* vérifie que les trajectoires générées soient correctes. Pour cela, une trajectoire est générée avec le modèle et le bloc. Les résultats sont ensuite comparés. Toutes les combinaisons des paramètres sont utilisées afin de s'assurer que toutes les trajectoires soient correctes.

Les fichiers spécifiques à ce banc sont disponibles dans le dossier :

`dev/01_trajectory_generation/src_tb/traj_calculator/`

***Pilote***

Deux pilotes, `BrakeDriver` et `CalculatorDriver`, sont utilisés pour ce test permettant de contrôler les différents signaux d'entrée du bloc.

Le premier simule la lecture de la LUT contenant le couple du frein magnétique. Les valeurs du couple sont mises à jour en fonction de l'index demandé par l'entité. Le signal indiquant que le frein ne doit pas être utilisé est également stimulé.

Le pilote `CalculatorDriver` effectue la séquence permettant de lancer le calcul de trajectoire. La séquence commence par modifier les paramètres de la trajectoire, puis démarre la génération et finalement attend que l'un des signaux indiquant la validité ou non de la trajectoire soit actif. Une action est notifiée à l'environnement de simulation au démarrage et à la fin de la génération.

***Moniteur***

Le moniteur `CalculatorMonitor` est utilisé pour lire les données provenant du calcul de trajectoire. Dès que le signal pour démarrer une nouvelle génération est activé, le moniteur va transmettre, via une file d'attente, la vitesse, position et courant à

chaque fois que l'écriture dans le mémoire est activée. Les signaux de validité et d'invalidité sont également transmis s'ils sont actifs, auquel cas le moniteur considère la génération comme terminée et arrête de transmettre les données jusqu'à la prochaine. Une action est notifiée à l'environnement de simulation à chaque fois qu'une donnée est lue.

### *Scoreboard*

Le scoreboard `CalculatorScoreboard` est utilisé pour valider la trajectoire générée. Celui-ci utilise deux instances de `CalculatorBlockScoreboard` pour les directions d'aller et de retour. Dans le cas de ce test, le DUT est directement le bloc *Trajectory calculator*, il n'y a donc pas besoin de la deuxième instance qui est utilisée pour le test de l'entité principale.

Afin de réduire les différences entre les données générées par le modèle et le DUT, le modèle est arrondi à la même précision que les sorties du DUT, soit $2^{-16}$ grâce à la classe `TrajectoryData`. Les trajectoires DUT n'étant pas exactement les mêmes que celles de références, dû à la précision des calculs, les comparaisons sont faites avec une certaine tolérance à l'aide de la fonction `math.isclose`[2] permettant de comparer deux nombres à virgule flottante avec une tolérance absolue et relative.

La figure 5.3 montre la différence de vitesse et de position entre le modèle et le DUT lorsque la précision des nombres à virgule fixe n'était pas totalement définie avec une vitesse cible de $10\,\mathrm{rad\,s^{-1}}$. La différence de vitesse est relativement plus élevée lors du freinage. Cela était dû au manque de précision de certain calcul qui accumulait une erreur et décalait le passage à l'étape de décélération dans le temps. Pour réduire l'impact de cet écart, la lecture de la trajectoire de référence est automatiquement décalée du nombre d'échantillons nécessaires afin de garder les vitesses proches de celles du DUT. Si ce décalage devient trop grand, une erreur est levée. Cependant, la précision des nombres à virgule a été modifiée après cette implémentation et ce décalage n'apparait plus. Le principe est cependant toujours présent dans le scoreboard, pour les cas où la précision serait à nouveau modifiée.

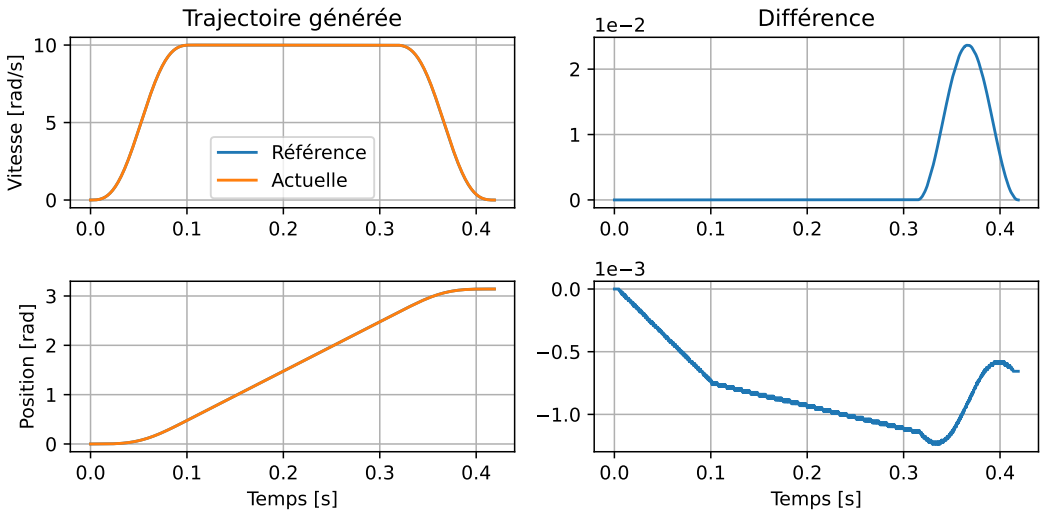


**Figure 5.3** – *Différence de vitesse élevée entre le modèle et le DUT*

2. https://docs.python.org/3/library/math.html#math.isclose

Lorsque le DUT a terminé la génération, le scoreboard vérifie que les signaux de validité et d'invalidité soit corrects. Si la génération est valide, les valeurs finales de position, vitesse et courant sont également comparées avec celles de la référence pour s'assurer que les calculs ne se soient pas arrêtés au milieu de la trajectoire. Le nombre de données générées est également vérifié et doit correspondre à celui de la référence avec une tolérance égale au décalage des échantillons maximal autorisé.

Finalement, toutes les valeurs des trajectoires, qui ont été comparées, sont écrites dans un fichier CSV avec le nom de la trajectoire, du profil de courant et de si le frein est pris en compte ou non. Un script Python permet d'afficher les graphiques pour visualiser les résultats et un deuxième pour les convertir en PDF. Ces scripts prennent en paramètre une liste de fichier CSV. Le deuxième prend, en plus, comme premier paramètre, le dossier de destination des PDFs. Ils sont disponibles dans les fichiers suivants :

`dev/01_trajectory_generation/script/csv_to_graph.py`

`dev/01_trajectory_generation/script/csv_to_pdf.py`

### 5.3.4 Test de l'entité principale

Le test de l'entité principale permet de vérifier que les deux blocs fonctionnent correctement ensemble. Pour cela, une trajectoire est d'abord générée comme dans le test de l'entité *Trajectory calculator*. Ensuite, les mouvements aller et retour sont simulés avec la trajectoire générée. Toutes les combinaisons des paramètres sont utilisées afin de s'assurer que toutes les trajectoires soient correctes.

Ce banc de test réutilise donc les pilotes, moniteurs et scoreboards des deux précédents, à l'exception du pilote `TrajectoryRamDriver` qui n'est pas utilisé, car il n'y a pas besoin de simuler la lecture depuis la RAM. Le moniteur `CalculatorMonitor` est utilisé deux fois pour les deux directions. Les suffixes `_in` et `_out` sont ajoutés aux noms des fichiers CSV générés par les scoreboards pour différencier les données de l'aller et du retour.

De plus le moniteur `TrajectoryRamMonitor` est utilisé. Celui-ci simule l'écriture dans la mémoire lors de la génération d'une trajectoire. Ces données sont stockées dans la classe `TrajectoryData`. Ainsi, à chaque fois que le signal activant l'écriture est actif, la donnée est écrite à l'index donné. Cela permet d'avoir les valeurs de référence exactes provenant du DUT lorsque le bloc *Trajectory lookup* est testé.

Les fichiers spécifiques à ce banc sont disponibles dans le dossier :

`dev/01_trajectory_generation/src_tb/traj_generation_top/`

## 5.4 Résultats

Tous les bancs de test ont été exécutés avec succès. Les figures 5.4 et 5.5 montrent deux comparaisons des trajectoires générées par le modèle et le DUT avec le test de l'entité principale.

En analysant les signaux avec l'outil graphique de QuestaSim, il est possible de déterminer qu'il faut 32 cycles d'horloge pour générer un échantillon de la trajectoire lors des phases d'accélération et de décélération. Il y a environ 2480 cycles d'horloge dans une période de la PWM, ouvrant la possibilité de générer les trajectoires en temps réel lors des mouvements, libérant ainsi les ressources mémoires utilisées par les trajectoires précalculées.

Les logs des tests, fichiers CSV et graphiques générés à partir de ceux-ci sont disponibles dans le dossier :

`dev/02_simulation_results/`

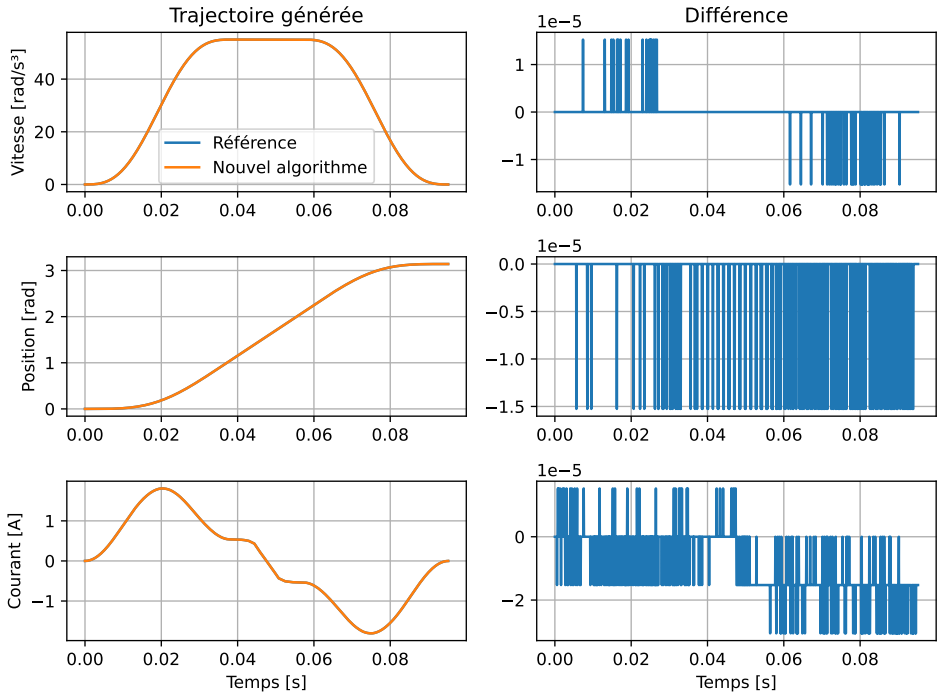


**Figure 5.4** – *Résultat de la génération* in *pour :* 55_pi *et* ONE_MILI *avec frein*

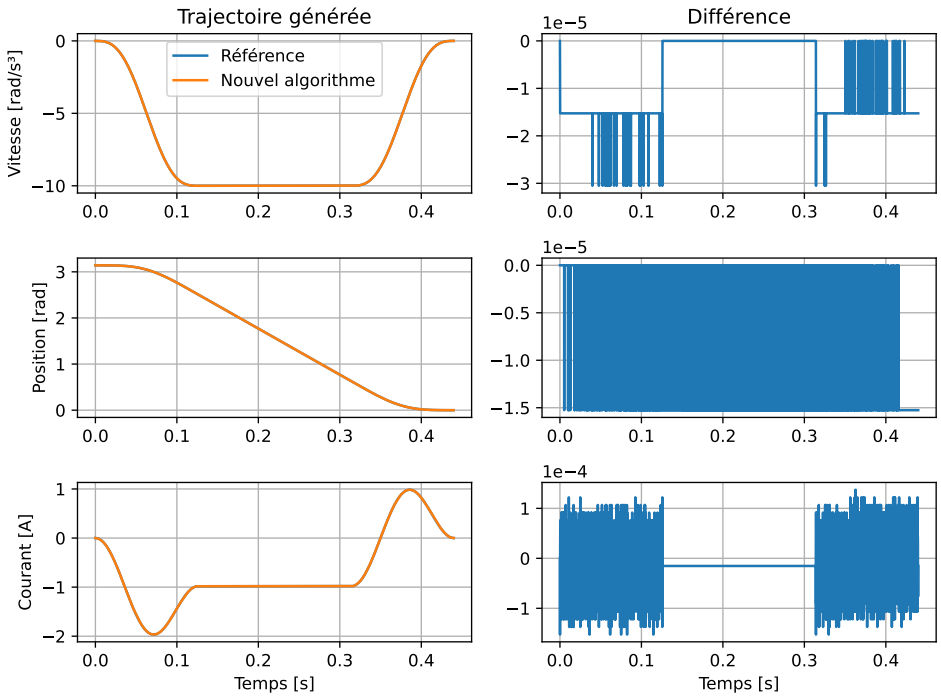


**Figure 5.5** – *Résultat de la génération* out *pour :* 10_pi *et* PS_LIU_S54 *sans frein*

# Chapitre 6

# Conclusion

L'objectif du projet est atteint, les trajectoires peuvent être générées directement sur la FPGA et, ainsi, un seul binaire est nécessaire. Une analyse de l'algorithme actuellement utilisé a permis de l'optimiser, ce qui fait gagner beaucoup de temps et réduit le nombre d'opérations nécessaires à la génération. De plus, une notion de trajectoire invalide est ajoutée afin d'assurer que les paramètres d'entrée ne puissent pas abîmer le scanner. Des bancs de test automatiques ont également été écrits en Python. Ces tests permettant de valider que les trajectoires générées soient correctes en les comparant à une référence et que les différentes entités VHDL fonctionnent correctement.

Les résultats obtenus sont positifs et le temps pris par la génération ouvre la possibilité de la faire en temps réel, soit directement lorsque le moteur est en mouvement, au lieu de pré-calculer les données. Cela n'était cependant pas une contrainte pour ce projet et n'a donc pas été considéré.

Un effort particulier a été mis pour rendre l'implémentation VHDL et les tests les plus génériques, compréhensibles et maintenables possibles. Les bancs de test pourraient encore être améliorés, par exemple, en ajoutant la possibilité de pouvoir choisir entre plusieurs cas différents à l'aide de paramètre dans la commande. Leur implémentation actuelle permet déjà de valider le fonctionnement de la description VHDL.

Des problèmes dans les scripts actuellement utilisés ont également pu être identifiés et corrigés. En effet, les tables de valeur étaient limitées à 4096 échantillons, ce qui n'était pas suffisant pour les trajectoires avec des vitesses basses qui peuvent aller jusqu'à 7000 échantillons. La trajectoire *out* était également fausse pour les trajectoires asymétriques. Celle-ci était reprise à partir de la trajectoire *in* en lisant les positions depuis la fin. L'opposé de la vitesse et du courant étaient utilisés, mais sans lire le tableau à l'envers, ces valeurs ne correspondaient donc pas aux positions. Ces problèmes ont été corrigés dans la nouvelle implémentation.

Un point qui manque dans la réalisation de ce projet est l'intégration de celui-ci dans le banc de test réalisé lors du projet de master précédent. Celui-ci consistait à vérifier le fonctionnement du régulateur utilisé par le *Beam Wire-Scanner*. En effet, après une première analyse des modifications à apporter à ces tests, le temps restant ne semblait pas suffisant et l'ajout des limites physiques sur les trajectoires ainsi que des améliorations des tests ont été faites à la place.

La planification prévue, disponible en annexe B, n'a pas été totalement respectée. L'analyse de l'algorithme a pris plus de temps que prévu, car il a fallu l'optimiser. L'interpolation pour le frein magnétique n'était pas prévue non plus. L'adaptation de l'algorithme a également déjà commencé à être faite durant l'optimisation. Les schémas ont dû être en partie refait après l'implémentation, car des modifications ont été faites dans l'implémentation.

D'un point de vue personnel, j'ai trouvé ce projet intéressant à réaliser. Cela m'a permis de mettre en pratique et renforcer mes connaissances en VHDL et sur l'utilisation de l'outil Quartus pour générer des IP Core. J'ai eu un peu plus de mal à faire les tests, n'ayant jamais utilisé Cocotb avant, il m'a fallu apprendre comment utiliser la librairie. L'avantage est la ressemblance avec SystemVerilog qui permet de ne pas partir de zéro. Ça a également été encourageant de réaliser un projet qui sera utilisé au CERN et j'espère que le résultat pourra être intégré lors des prochaines maintenances.

Je tiens à remercier M. Yann Thoma pour son suivi tout au long du projet. Ses conseils ont permis de mener ce projet à bien, notamment avec l'utilisation des IP Core ou de Cocotb.

Je remercie également M. Jonathan Emery du CERN pour sa disponibilité malgré des obligations professionnelles plutôt chargées durant ce projet. Les différentes informations et réponses qu'il m'a données m'ont permis de faire les meilleurs choix.

Clément Dieperink

# Bibliographie


[BM12] Luigi Biagiotti et Claudio Melchiorri. « FIR filters for online trajectory planning with time- and frequency-domain specifications ». In : *Control Engineering Practice* 20.12 (2012), p. 1385-1399. ISSN : 0967-0661. DOI : https://doi.org/10.1016/j.conengprac.2012.08.005. URL : https://www.sciencedirect.com/science/article/pii/S0967066112001669 (cf. p. 11).

[Eme+19] Jonathan Emery et al. « A low fluctuation control strategy for PMSM direct drive system targeting Particle Beam Instrumentation Application ». In : (2019), p. 437-443. DOI : 10.1109/CCTA.2019.8920688 (cf. p. 7, 12).

[BM21] Luigi Biagiotti et Claudio Melchiorri. « Optimization of Generalized S-Curve Trajectories for Residual Vibration Suppression and Compliance With Kinematic Bounds ». In : *IEEE/ASME Transactions on Mechatronics* 26.5 (2021), p. 2724-2734. DOI : 10.1109/TMECH.2020.3045504 (cf. p. 10, 11).

# Annexes

# Annexe A

# Documentation d'utilisation

## *A.1 Génération d'une trajectoire*

La séquence pour utiliser le générateur est la suivante :

1. Mettre à jour tous les paramètres de la trajectoire
2. Activer le signal `start_generation_i` pendant un cycle d'horloge
3. Attendre quatre cycles d'horloge pour assurer la transmission des signaux
4. Attendre que `trajectory_valid_o` ou `parameters_invalid_o` soit actif

En tout temps, les signaux `brake_lut_value_i` et `brake_lut_value_2_i` doivent être mis à jour avec les valeurs de la LUT du frein magnétique en fonction des index donnés par les signaux `brake_lut_index_o` et `brake_lut_index_2_o`. Un délai maximum de trois cycles d'horloge doit être respecté entre la mise à jour de l'index et la mise à jour de la valeur. Si le frein n'est pas utilisé, les valeurs des signaux `brake_lut_value_i` et `brake_lut_value_2_i` n'ont pas d'importance.

Les signaux d'entrées et de sorties respectent les types et tailles donnés dans la table A.1. La notation `[partie entière].[partie fractionnaire]` est utilisées pour indiquer la taille des nombres à virgule fixe.

**Table A.1** – *Types des signaux d'entrée et sortie pour la génération*

| Signal | Type | Nombre de bits |
|---|---|---|
| `starting_position_i` | Virgule fixe signé | 8.24 |
| `spd_target_i` | Virgule fixe signé | 16.16 |
| `dist_acc_i` | Virgule fixe signé | 8.24 |
| `dist_dec_i` | Virgule fixe signé | 8.24 |
| `dist_total_i` | Virgule fixe signé | 8.24 |
| `inertia_i` | Virgule fixe signé | 4.28 |
| `dyn_fric_factor_i` | Virgule fixe signé | 4.28 |
| `sta_fric_i` | Virgule fixe signé | 4.28 |
| `torque_cst_i` | Virgule fixe signé | 4.28 |
| `no_brake_i` | Booléen | 1 |
| `brake_lut_value_i` | Virgule fixe signé | 4.28 |
| `brake_lut_value_2_i` | Virgule fixe signé | 4.28 |
| `brake_lut_index_o` | Entier non signé | 9 |
| `brake_lut_index_2_o` | Entier non signé | 9 |

## A.2 Lecture de la trajectoire

La figure A.1 montre la séquence pour lire la trajectoire. Le nombre de cycles d'horloge entre chaque activation du signal `increment_pointer_by_one_i` est libre. La génération de la trajectoire doit être valide pour que la lecture soit faite.

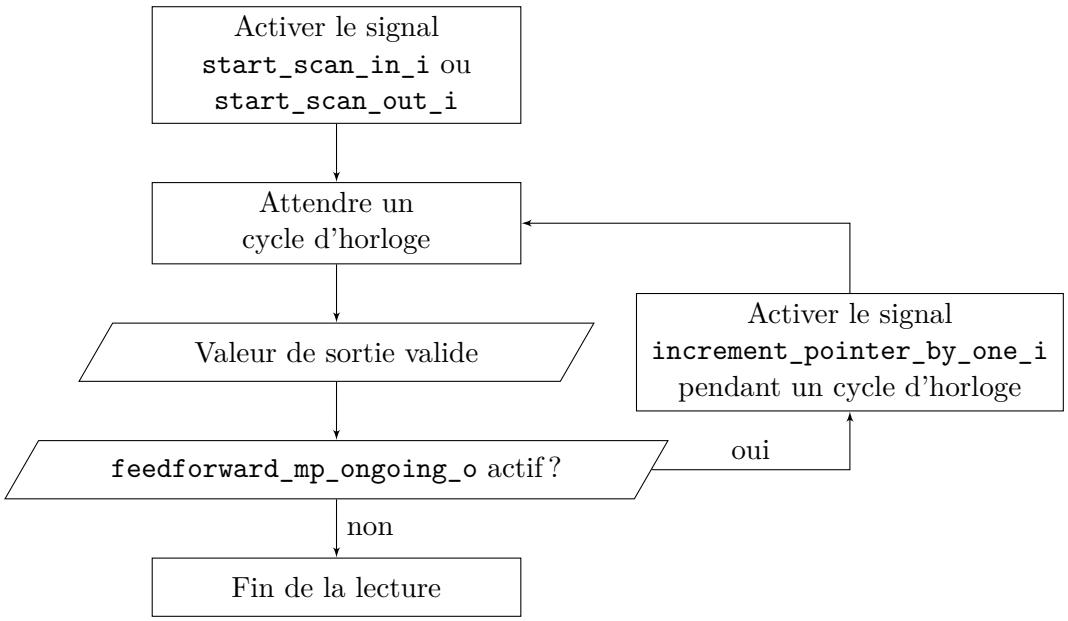


**Figure A.1** – *Lecture de la trajectoire*

Les signaux d'entrées et de sorties respectent les types et tailles donnés dans la table A.2. La notation `[partie entière].[partie fractionnaire]` est utilisées pour indiquer la taille des nombres à virgule fixe.

**Table A.2** – *Types des signaux d'entrée et sortie pour la lecture*

| Signal | Type | Nombre de bits |
|---|---|---|
| `increment_pointer_by_one_i` | Booléen | 1 |
| `start_scan_in_i` | Booléen | 1 |
| `start_scan_out_i` | Booléen | 1 |
| `feedforward_mp_ongoing_o` | Booléen | 1 |
| `feedforward_mp_ongoing_in_o` | Booléen | 1 |
| `feedforward_mp_ongoing_out_o` | Booléen | 1 |
| `feedforward_mp_cur_o` | Virgule fixe signé | 16.16 |
| `feedforward_mp_spd_o` | Virgule fixe signé | 16.16 |
| `feedforward_mp_pos_o` | Virgule fixe signé | 16.16 |

# Annexe B

# Planification

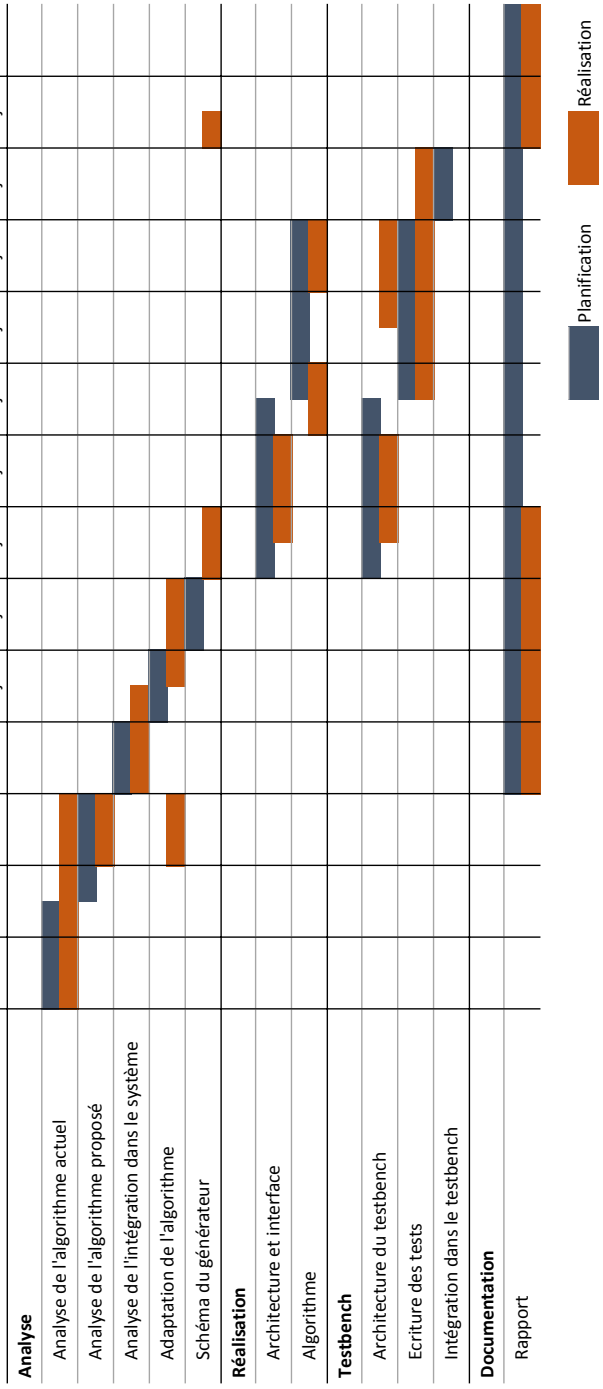


**Figure B.1** – *Planification*

# Annexe C

# Comparaison des algorithmes

Cette annexe présente plusieurs graphes de comparaison des trajectoires générées avec l'algorithme de référence et l'algorithme optimisé. Les paramètres utilisés sont repris de la génération de trajectoire actuelle.

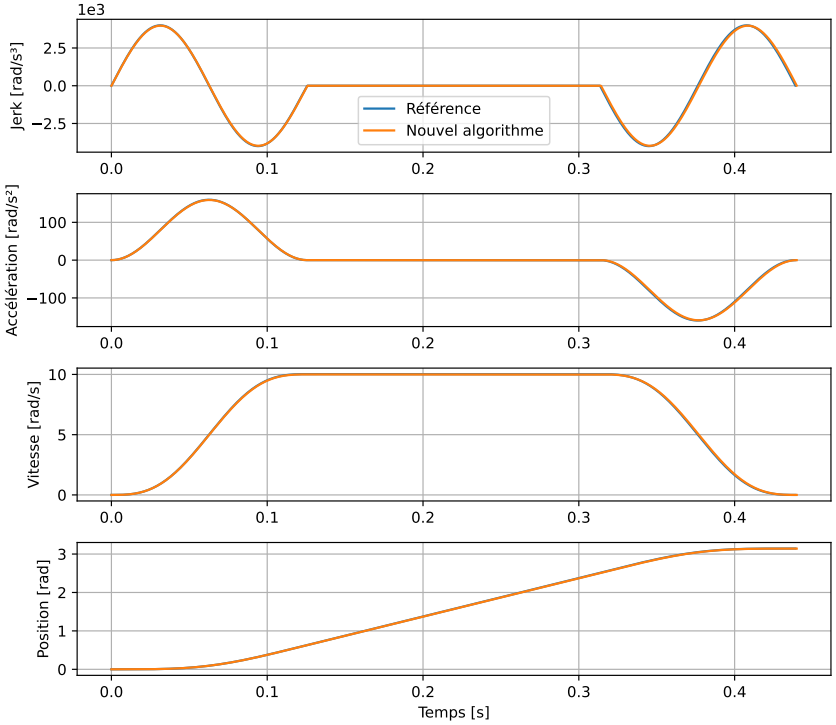


**Figure C.1** – *Comparaison de la trajectoire avec l'algorithme de référence et la nouvelle version,* $\omega_{target} = 10\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{5}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

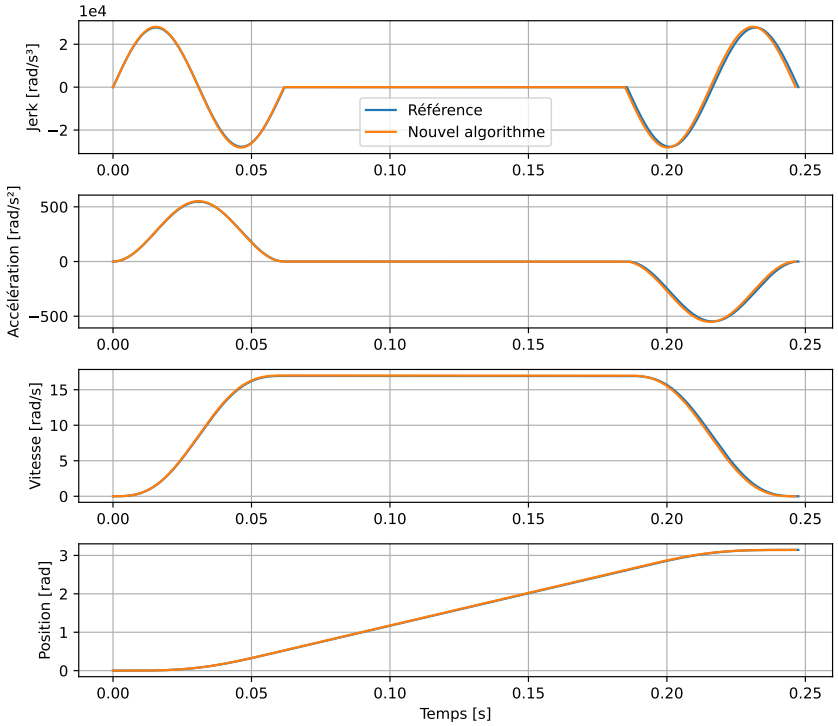


**Figure C.2** – *Comparaison de la trajectoire avec l'algorithme de référence et la nouvelle version,* $\omega_{target} = 17\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{6}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

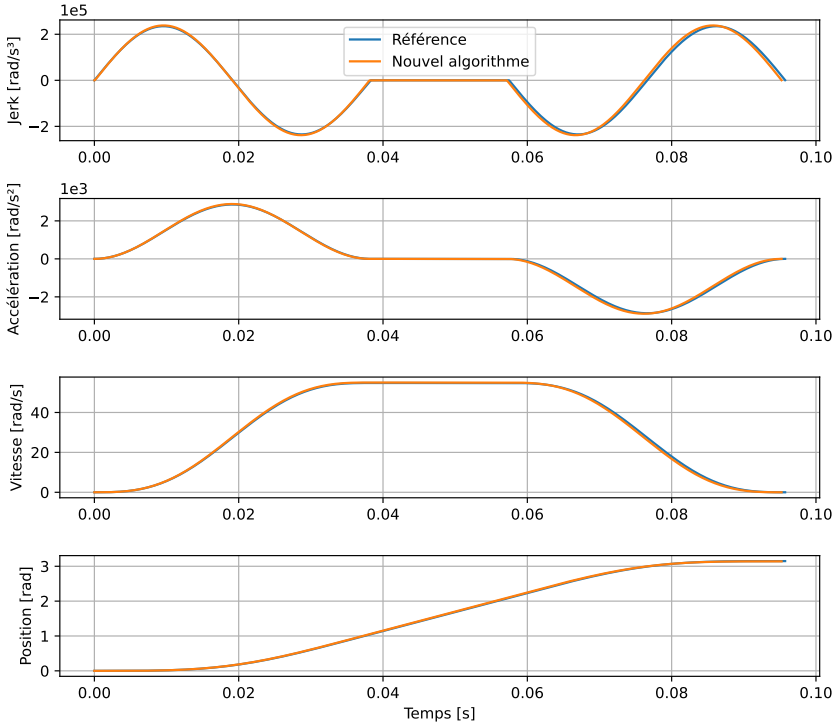


**Figure C.3** – *Comparaison de la trajectoire avec l'algorithme de référence et la nouvelle version,* $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{3}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

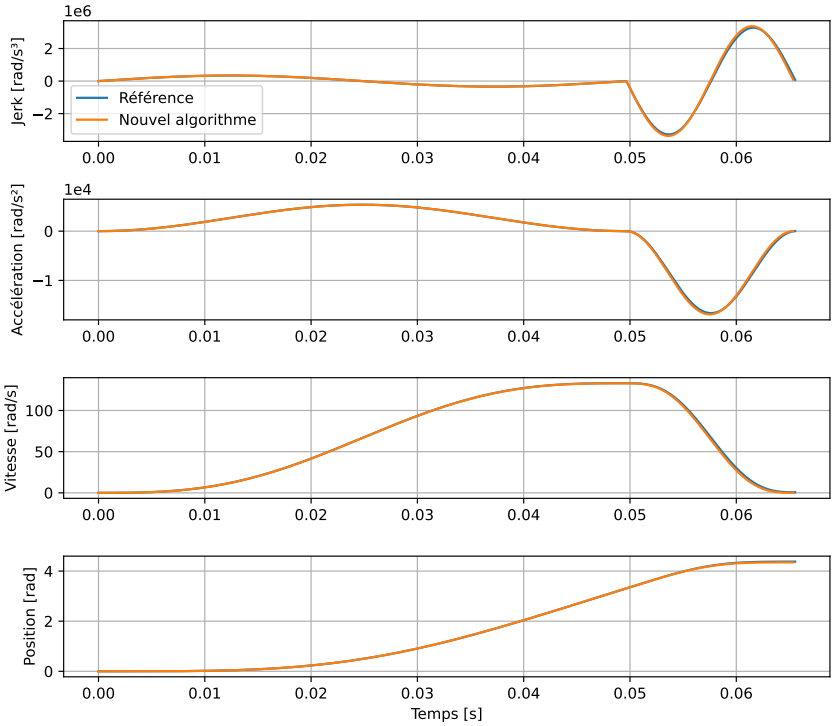


**Figure C.4** – *Comparaison de la trajectoire avec l'algorithme de référence et la nouvelle version,* $\omega_{target} = 133\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = 3.298\,\mathrm{rad}$, $\theta_{dec} = \frac{\pi}{3}\,\mathrm{rad}$ *et* $\theta_{total} = 1.38\pi\,\mathrm{rad}$

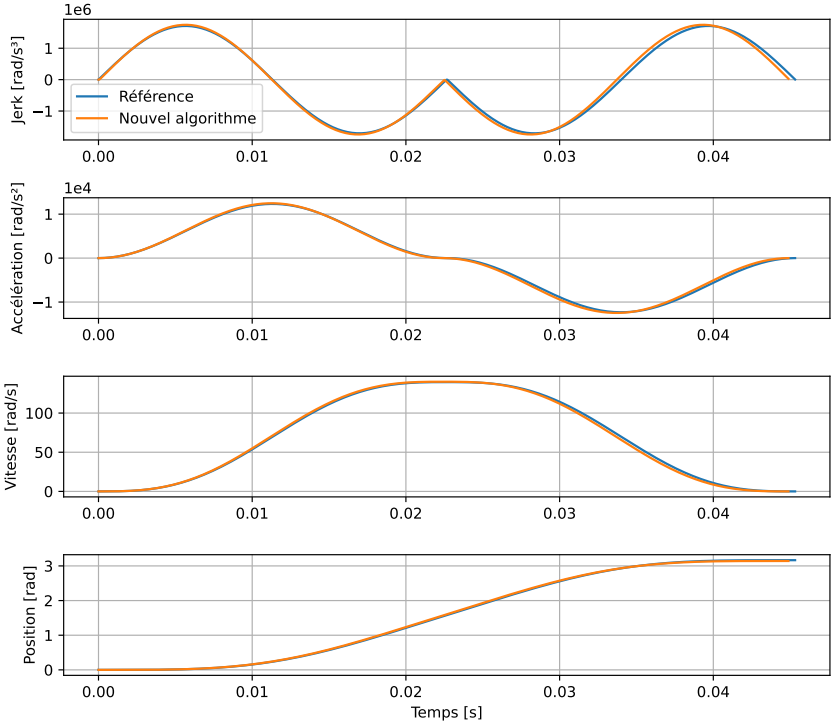


**Figure C.5** – *Comparaison de la trajectoire avec l'algorithme de référence et la nouvelle version,* $\omega_{target} = 140\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{2}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

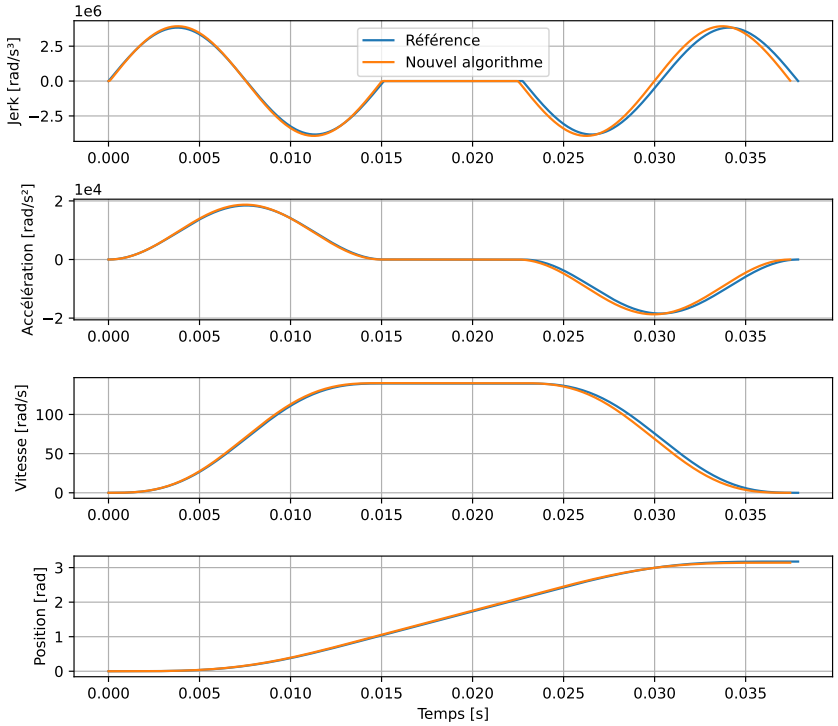


**Figure C.6** – *Comparaison de la trajectoire avec l'algorithme de référence et la nouvelle version,* $\omega_{target} = 140\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{3}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

# Annexe D

# Comparaison des types de variables

Cette annexe présente plusieurs graphes de comparaison des trajectoires générées avec des nombres à virgule flottante et des nombres à virgules fixes sur 36 bits. Les paramètres utilisés sont repris de la génération de trajectoire actuelle.

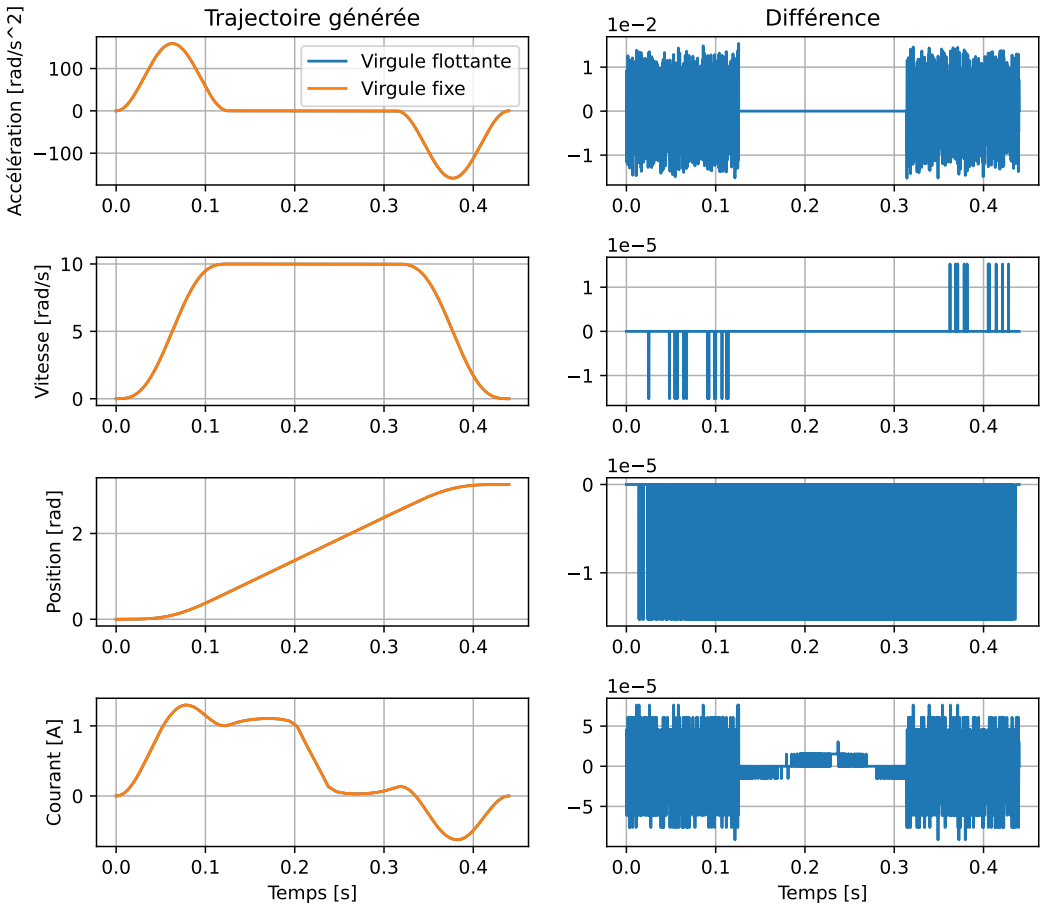


**Figure D.1** – *Comparaison de la trajectoire avec des nombres à virgule flottante et des nombres à virgule fixe,* $\omega_{target} = 10\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{5}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

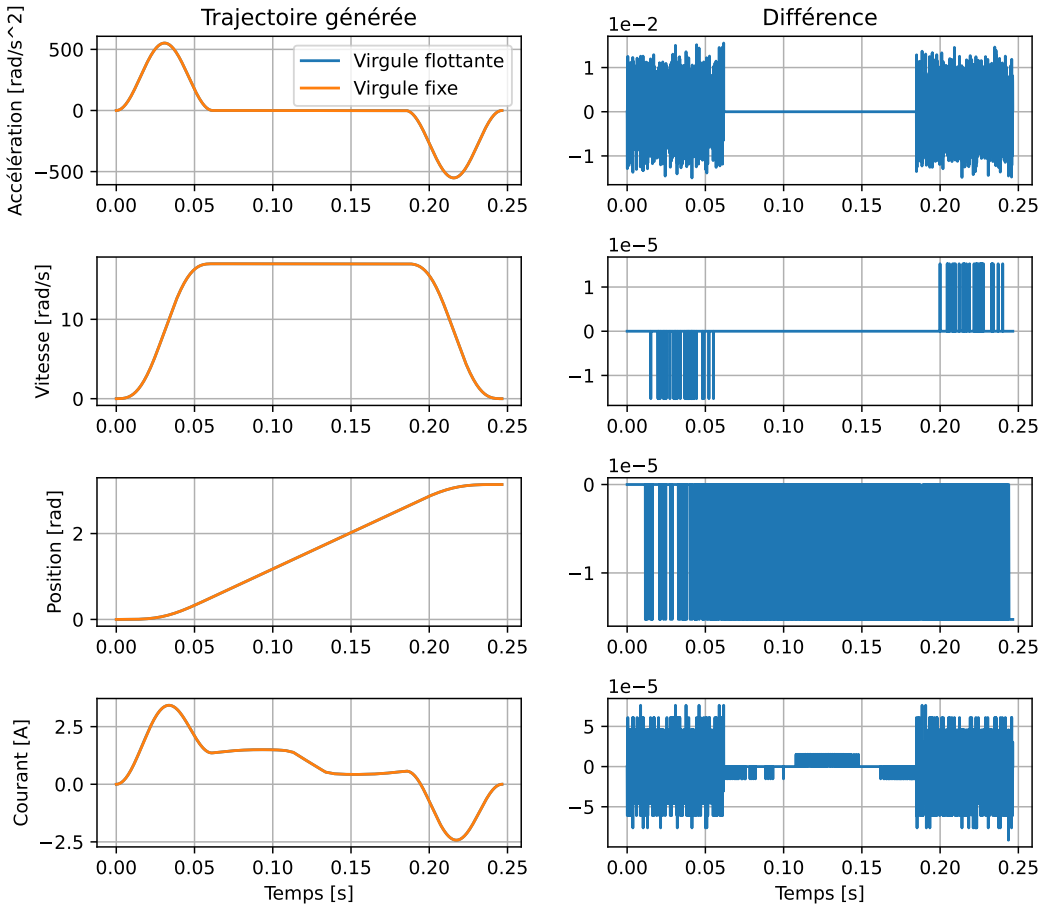


**Figure D.2** – *Comparaison de la trajectoire avec des nombres à virgule flottante et des nombres à virgule fixe,* $\omega_{target} = 17\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{6}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

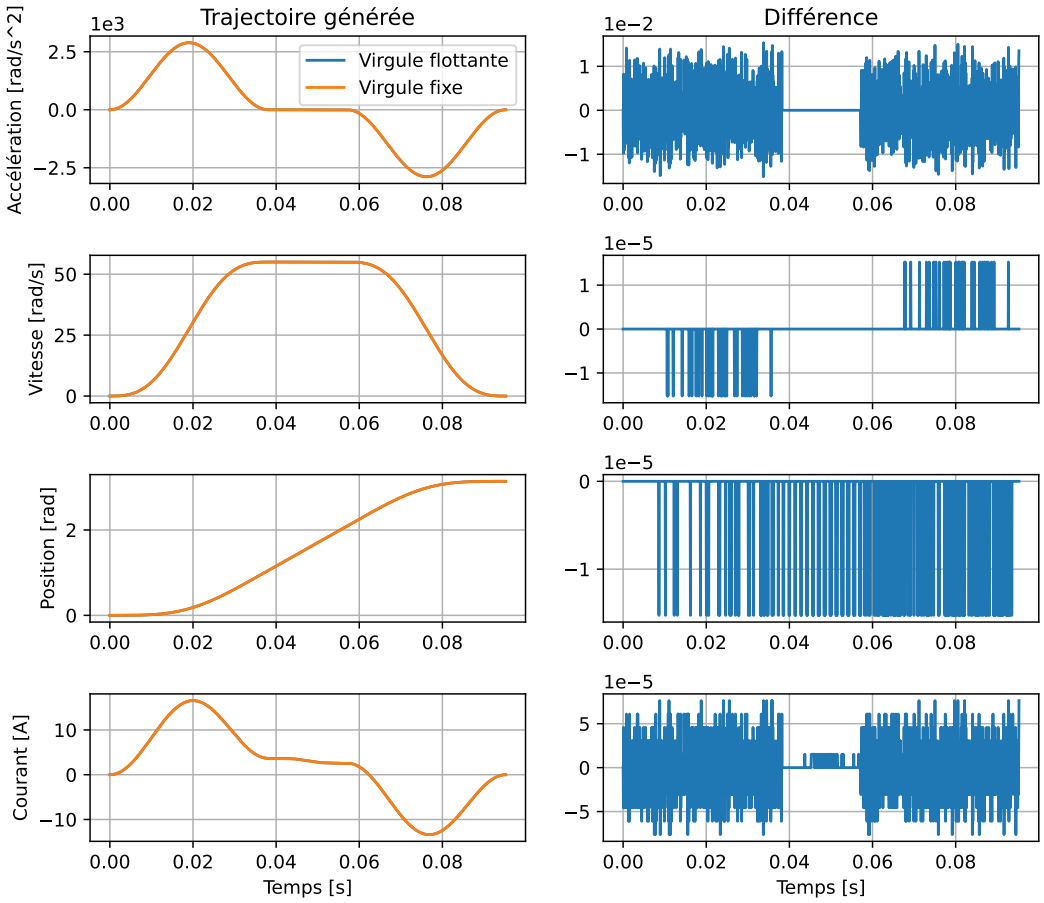


**Figure D.3** – *Comparaison de la trajectoire avec des nombres à virgule flottante et des nombres à virgule fixe,* $\omega_{target} = 55\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{3}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

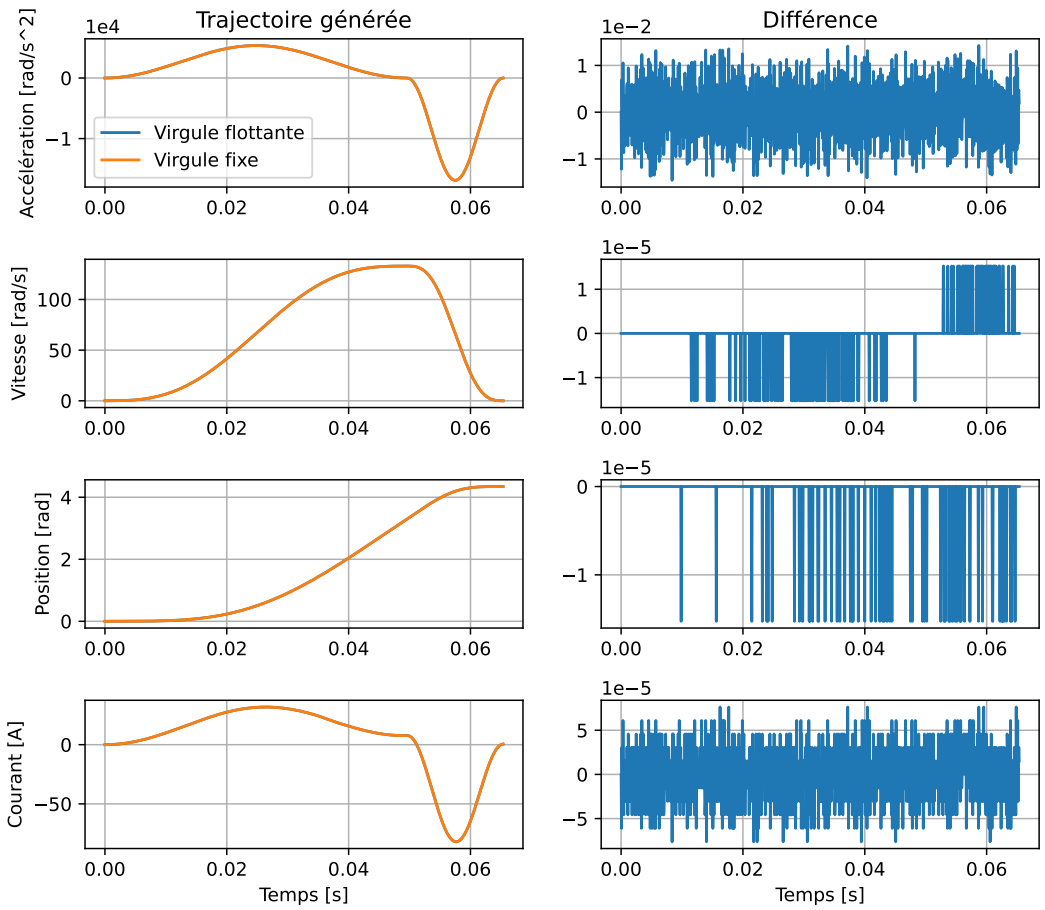


**Figure D.4** – *Comparaison de la trajectoire avec des nombres à virgule flottante et des nombres à virgule fixe,* $\omega_{target} = 133\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = 3.298\,\mathrm{rad}$, $\theta_{dec} = \frac{\pi}{3}\,\mathrm{rad}$ *et* $\theta_{total} = 1.38\pi\,\mathrm{rad}$

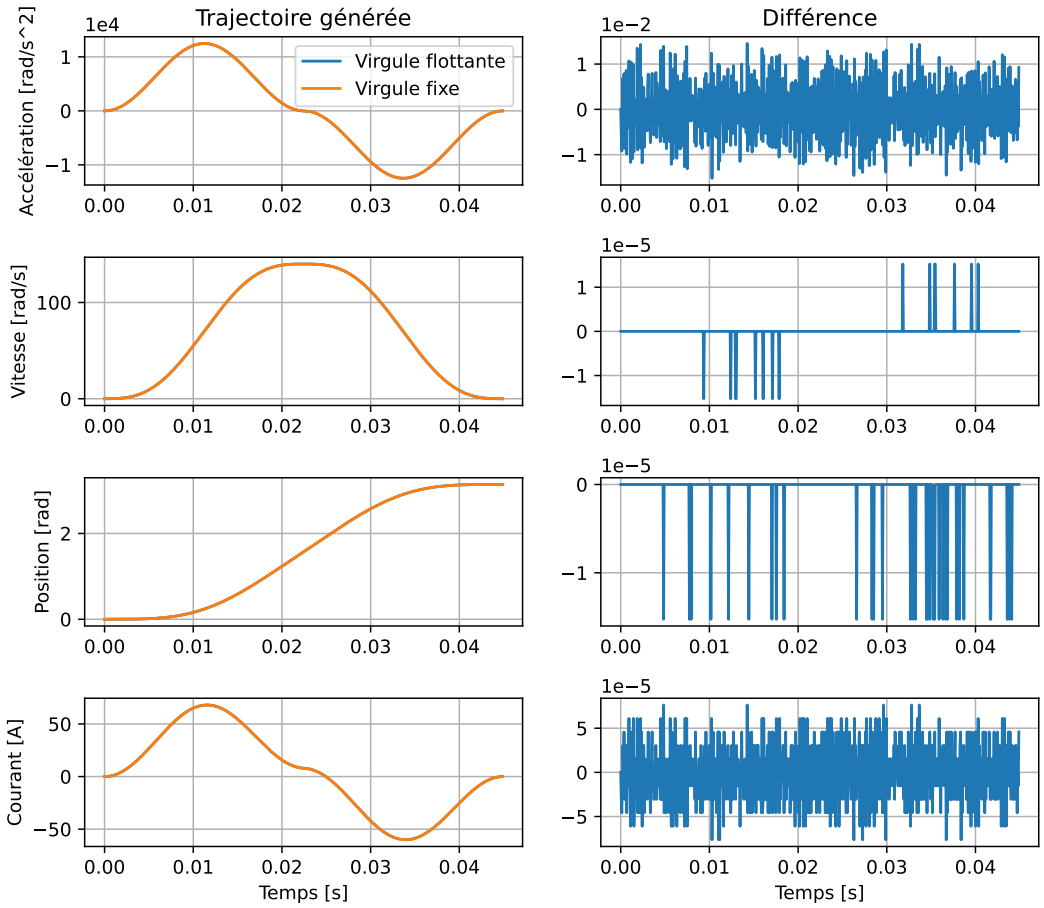


**Figure D.5** – *Comparaison de la trajectoire avec des nombres à virgule flottante et des nombres à virgule fixe,* $\omega_{target} = 140\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{2}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

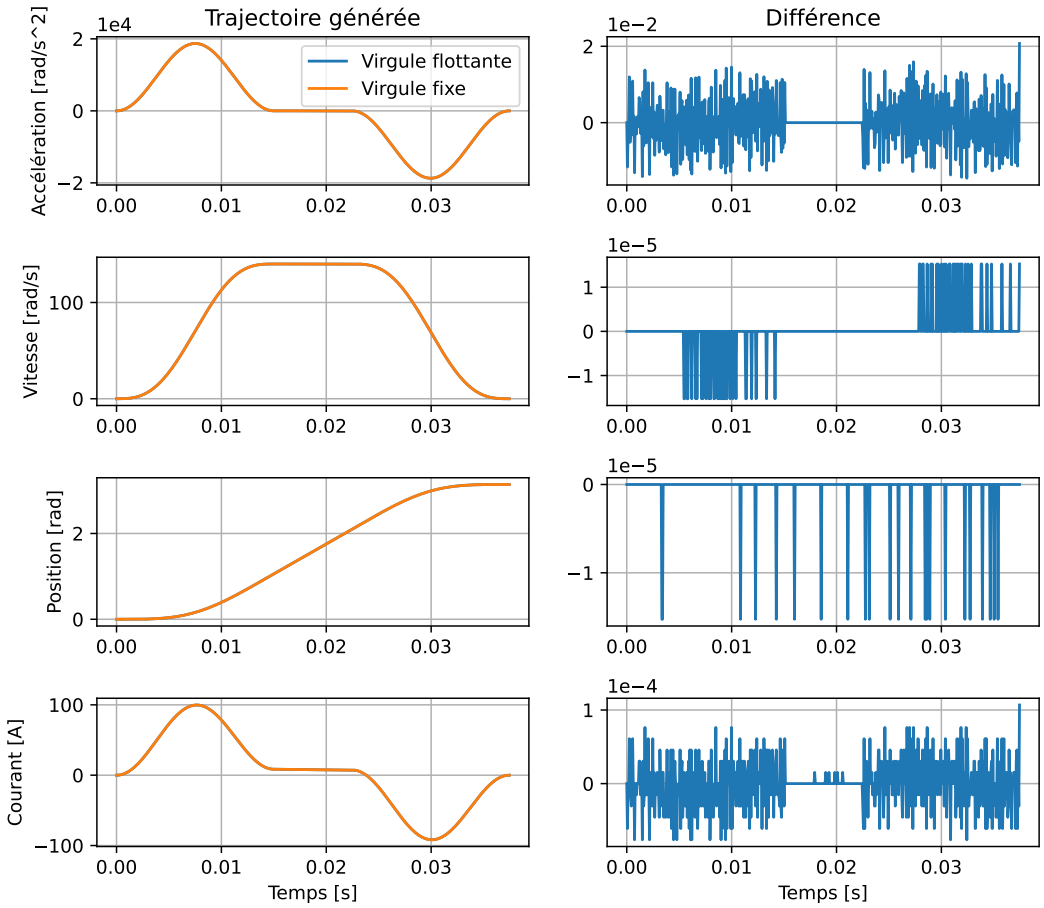


**Figure D.6** – *Comparaison de la trajectoire avec des nombres à virgule flottante et des nombres à virgule fixe,* $\omega_{target} = 140\,\mathrm{rad\,s^{-1}}$, $\theta_{acc} = \theta_{dec} = \frac{\pi}{3}\,\mathrm{rad}$ *et* $\theta_{total} = \pi\,\mathrm{rad}$

# Glossaire

**BWS** *Beam Wire-Scanner*. v, ix, 1, 6, 7, 8, 43

**CERN** Organisation Européenne pour la Recherche Nucléaire. 1, 10, 44

**DSP** *Digital Signal Processor*, processeur de signal numérique permettant de faire des opérations mathématiques sur des signaux. 33, 34

**DUT** *Device Under Test*, composant à tester. 35, 36, 37, 40, 41

**FPGA** *Field-programmable gate array*, Circuit logique programmable. 1, 2, 5, 9, 12, 13, 30, 31, 33, 43

**IP Core** *Intellectual Property Core*, unité logique prête à être utilisée. 24, 30, 31, 34, 44

**LUT** *Look Up Table*, table de correspondance entre une entrée et une sortie. 7, 13, 21, 25, 28, 32, 39, 49

**PWM** *Pulse Width Modulation*, modulation de largeur d'impulsion, permet de générer un signal analogique à partir d'un signal numérique. 6, 11, 12, 14, 22, 25, 28, 29, 38, 41

**RAM** *Random Access Memory*, mémoire permettant de stocker des données en lecture et écriture. 21, 23, 24, 29, 41

**ROM** *Read Only Memory*, mémoire permettant de stocker des données en lecture seule. 29

**VHDL** Langage de description de matérielle. xi, 2, 3, 8, 27, 28, 33, 35, 43, 44