Modélisation et simulation robotique

MU4RBR05 - S2-23

Compte rendu du projet

Auteur:

Nemanja Maksimovic (28706774) nemanja.maksimovic@etu.sorbonne-universite.fr

M1 SAR

2023/2024 – 2nd Semestre

Contents

1 Introduction

Le but de se projet est de mettre en oeuvre des principes de simulation physique vu en cours dans le contexte de programmation orientée objet en Python. Plusieurs sujets vont être traité. Nous allons apporter nos solutions à chaque sujet en détaillant les choix qui ont ont pu être fait et montrer des exemples des implémentations.

2 Simulation d'un moteur à courant continu (CC)

L'objectif était de créer un modèle numérique du moteur à courant continu. Ce moteur est représenté par des grandeurs électriques :

R : résistance de l'induit **L** : inductance de l'induit
E(t) : force contre-électromotrice **Um(t)** : tension aux bornes du moteur
i(t) : courant

et par des grandeurs mécaniques :

J : inertie du rotor **f** : frottements visqueux Γ**(t)** : couple moteur
Ω**(t)** : vitesse du rotor

Le modèle se construit à partir des relations électriques et mécaniques suivantes :

$$\text{Le couple moteur : } \Gamma(t) = k_c i(t) \tag{1}$$

avec k_c : constante de couple

$$\text{La force contre-électromotrice : } E(t) = k_e \Omega(t) \tag{2}$$

avec k_e : constante contre-électromotrice

$$\text{l'équation électrique : } Um(t) = E(t) + Ri(t) + L\frac{di(t}{dt} \tag{3}$$

$$\text{l'équation mécanique : } J\frac{d\Omega(t)}{dt} + f\Omega(t) = \Gamma(t) \tag{4}$$

2.1 Solution analytique

Dans un premier temps, nous allons établir la solution analytique de Ω(t) en fonction de $U_m(t)$ avec la simplification $L \approx 0H$.

Avec (2) et (3) on a : $U_m(t) = k_e\Omega(t) + Ri(t)$
$\Leftrightarrow$ Ω(t) = $\frac{1}{ke}$(U$_m(t) - Ri(t)$)

avec (1) et (4) on obtient : $\Omega(t) + \tau\frac{d\Omega(t)}{dt} = KU_m$
avec $\tau = \frac{RJ}{kekc+Rf}$ et $K = \frac{kc}{kekc+Rf}$

On obtient une équation différentielle du premier ordre avec second membre.

solution homogène : $\Omega_1(\mathrm{t}) = \alpha \exp(-\frac{t}{\tau})$ et solution particulière : $\Omega_2(\mathrm{t}) = \mathrm{KU}_m$
avec les conditions initiales, on trouve comme solution à cette équation :

$$\Omega(t) = K(1 - \exp(-\frac{t}{\tau}))U_m \tag{5}$$

2.2 Simulation

Nous avons crée une classe 'MoteurCC' et nous l'avons intégré à un simulateur. Pour la simulation, nous allons considéré que $L \approx 0H$. Pour construire un objet de cette classe, nous allons lui attribuer les paramètres physiques : R, L, kc, ke, J et f. Pour enrichir notre modèle, nous avons également rajouté en paramètre d'entrée des actions extérieures : inertie de charge Jc, et un couple extérieur Γ_{ext}. Nou pouvons également définir la position du centre du moteur (*center*) dans l'espace, ce paramètre sera de type 'Vecteur3D' comme vu durant les TP.
Nous avons initialisé des attributs à partir des paramètres d'entrées. Nous avons aussi des listes qui vont enregistrer les valeurs de certaines grandeurs physiques qui vont être calculées au cours de la simulation : le courant, la vitesse angulaire, le couple moteur et la position angulaire du moteur.
Parmi les méthode, certaines vont permettre de récupérer les derniers éléments des listes (**getIntensity**, **getSpeed**, **getTorque** et **getPosition**). Une méthode **SetVoltage** va permettre d'envoyer une tension, qui sera défini en paramètre, à notre moteur. Enfin, une dernière méthode **simule** effectue une simulation du comportement du moteur pendant un intervalle de temps dt. Elle calcule la nouvelle vitesse angulaire, le nouveau courant, le nouveau couple moteur et met à jour la position en fonction des équations de la dynamique du moteur à courant continu. Voici comment sont effectuées les calculs:

- $\frac{d\Omega(t)}{dt} = \frac{kc}{R}U_m(t) - (\frac{kekc}{R} + f)\Omega(t-1)$, on a $\Omega(t) = \Omega(t-1) + \frac{d\Omega(t)}{dt} * dt$
- $i(t) = (U_m(t) - ke * \Omega(t))/R$
- $\Gamma(t) = \Gamma ext + i(t)kc$
- $Position(t) = Position(t-1) + \Omega(t)dt$

Pour vérifier notre système, nous allons tracer la réponse indicielle théorique en boucle ouverte (échelon unité de tension) avec notre simulateur et nous allons la comparer avec la réponse théorique. Pour cela, on utilisera les valeurs suivantes :

résistance de l'induit : 1Ω	inductance de l'induit : $0.001H \approx 0H$
constante de couple : 0, 01N m/A	constante de la fcem : 0, 01V.s
inertie du rotor : 0,01kg.m2	constante de frottement visqueux : 0,1N m.s

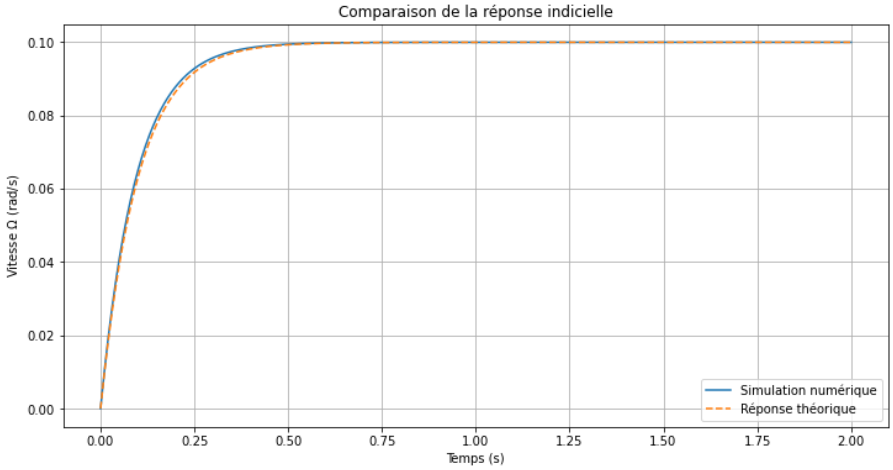


Figure 1: Réponse indicielle de MoteurCC en boucle ouverte

Pour notre simulation, nous avons imposé une tension de 1 V. Nous constatons sur la figure que la réponse indicielle de la simulation numérique est très semblable à la théorique. Nous pouvons valider notre modèle.
Pour faire le tracé de la réponse théorique, nous repronons l'équation (5), avec les valeurs données en entrée, on trouve $K \approx 0.1 rad/s/V$ et $\tau \approx 0.1 s$

2.3 Commande en vitesse

Nous avons insérez notre modèle dans un schéma de commande en boucle fermé pour la régulation de la vitesse. Pour cela, nous avons crée une classe 'ControlPID_vitesse' qui implémente un contrôleur PID pour réguler la vitesse d'un moteur à courant continu. Les paramètres du constructeur de cette classe vont être un objet de la classe 'MoteurCC', ainsi que Kp, Ki et Kd qui sont respectivement les gains proportionnel, intégral et dérivé du contrôleur PID. Parmi les attributs importants, nousa avons :

- target_speed : Vitesse cible que le contrôleur PID cherche à atteindre
- previous_error : Erreur de vitesse à l'itération précédente, utilisée pour le calcul du terme dérivé
- integral : Somme accumulée des erreurs, utilisée pour le calcul du terme intégral
- voltages : Liste des tensions appliquées au moteur au cours du temps

De même, pour les méthodes importantes nous avons :

- setTarget(vitesse): Définit la vitesse cible que le contrôleur PID doit atteindre
- getVoltage(): Retourne la dernière tension appliquée au moteur. Si aucune tension n'a été appliquée, retourne 0

- simule(dt): Simule le comportement du contrôleur PID et du moteur pendant un intervalle de temps dt. Elle calcule l'erreur actuelle, met à jour l'intégrale de l'erreur, calcule le terme dérivé, puis calcule et applique la tension nécessaire au moteur. La méthode met également à jour la liste des tensions appliquées

Dans la méthode **simule(dt)**, la tension à appliquer est calculée en utilisant la formule du contrôleur PID, qui combine les termes proportionnel, intégral et dérivé:
$U_m(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)\,d\tau + K_d \cdot \frac{de(t)}{dt}$
avec l'erreur : $e(t) = \text{vitesse\_cible} - \Omega(t)$

On va d'abord étudier le cas d'un correcteur proportionnel, soit pour Kd=Ki=0. Pour les valeurs du moteur, nous allons reprendre les mêmes valeurs que précédement.

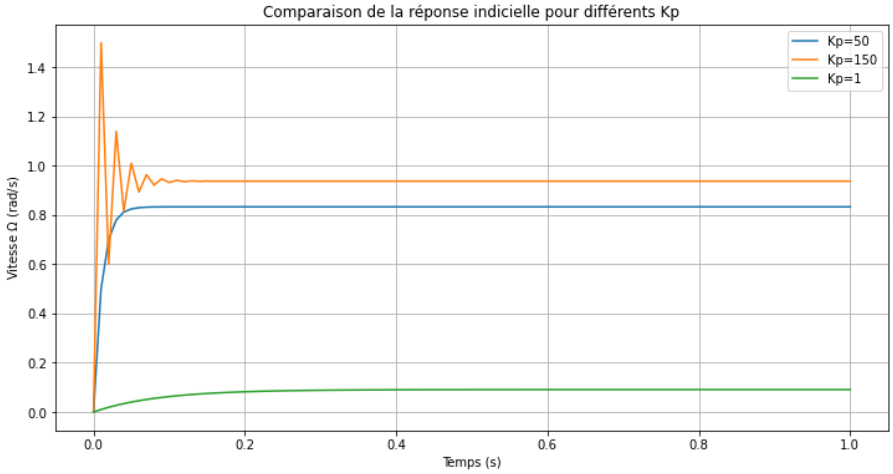


Figure 2: Réponse indicielle avec un correcteur proportionnelle

Pour la simulation, nous avons mis une vitesse cible de 1 rad/s. Nous pouvons constater sur la figure, le fait d'augmenter Kp va permettre d'avoir une réponse plus rapide et de se rapprocher davantage de la valeur cible. Aussi, un gain proportionnel plus élevé réduit l'erreur entre la consigne et la sortie. En revanche, nous pouvons constater que pour une valeur de Kp trop élevé, comme ici pour Kp=150, cela peut provoquer des oscillations et rendre notre système plus instable. Pour décider de quelle valeur de Kp est le plus optimal pour notre système, cela dépend si on privilégie une réponse qui est plus proche de la vitesse cible malgré des oscillations, ou alors une réponse plus stable malgré une certaines différence avec la vitesse cible. Dans le premier cas, on peut choisir une valeur de Kp dans les alentours de 180, ou dans le deuxième cas une valeur dans les alentours de 80.

Table 1: Caractéristiques du système pour certaines valeurs de Kp en régime permanent

Kp	temps de réponse	erreur statique	Ω	U_m	Γ	i
80	0.02 s	0.11	0.89 rad/s	8.90 V	0.09 Nm	8.89 A
180	0.3 s	0.05	0.95 rad/s	9.48 V	0.09 Nm	9.47 A

Nous allons à présent étudier le cas d'un correcteur proportionnel+intégrateur, pour cela on aura Ki=0 et on va fixer Kp=50.

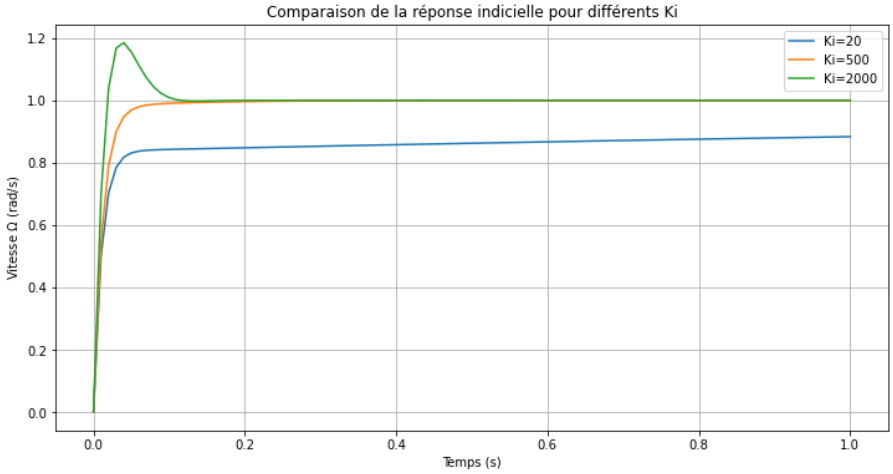


Figure 3: Réponse indicielle avec un correcteur proportionnelle+intégrateur

Le fait de rajouter un correcteur intégrateur va permettre d'éliminer l'erreur statique. En effet, le terme intégral va accumuler l'erreur au fil du temps et corriger toute erreur de décalage, assurant ainsi que l'erreur finale soit nulle. De plus, augmenter le gain intégral va permettre de réduire le temps de réponse. En revanche, un gain trop élevé peut provoquer des oscillations sur la réponse, comme nous pouvons nous en apercevoir pour Ki=2000. Ici, un bon choix de Ki serait une valeur aux alentours de 500.

Table 2: Caractéristiques du système pour Kp=50 et Ki=500

temps de réponse	erreur statique	Ω	Um	Γ	i
0.05 s	0.0	1.0 rad/s	10.0 V	0.1 Nm	10 A

En ce qui concerne le gain dérivé, il permet d'atténuer des oscillations et rendre notre système plus stable. En revanche, un gain élevé peut rendre le système plus sensible au bruit. Dans notre système, un gain adéquat serait dans les alentours de 0.2.

2.4 Commande en position

Nous allons explorer les mêmes principes que précédemment mais cette fois-ci pour une commande en position. Pour cela, une classe 'ControlPID_Position' a été crée. Cette classe possède les mêmes attributs et méthodes que 'ControlPID_Vitesse', seul l'interprétation sera différente. Par exemple, l'attribut **previous_error** correspond à l'erreur de position à l'itération précédente, ou dans la méthode **simule(dt)**, l'erreur est calculé en faisant la différence entre la position cible et la position obtenue. Nous allons étudier le cas d'un correcteur proportionnel.

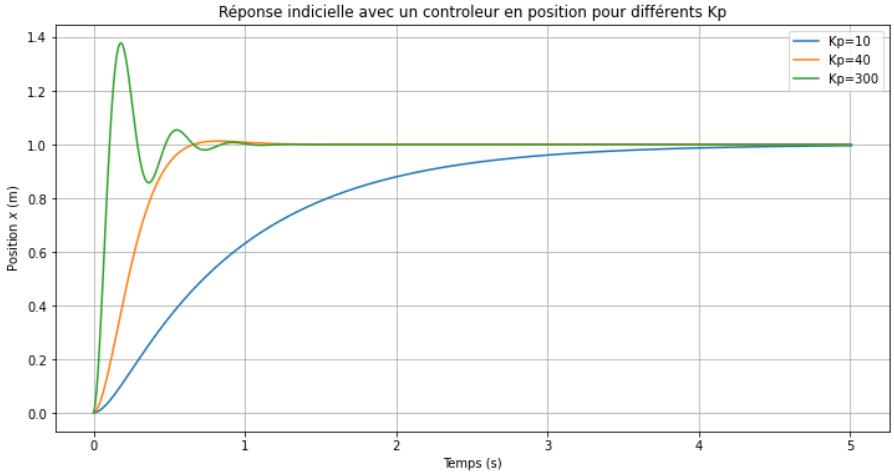


Figure 4: Réponse indicielle pour une commande en position avec un correcteur proportionnelle

Pour la simulation, on cible une position de 1 rad. Augmenter Kp va permettre d'avoir un temps de réponse plus réduite. En revanche, un gain trop élevé peut provoquer des oscillations sur notre système, comme nous pouvons le voir pour Kp=300. On constate par ailleurs que l'erreur statique est nul. Il semblerait alors qu'il soit inutile de rajouter un gain intégral, d'ailleurs, cela pourait provoquer plus d'oscillations sur la réponse.

3 Simulation [moteur + particule]

Dans ce nouveau problème, nous allons reprendre les codes faits en TP (Particule, Univers, Force...) et nous allons changer l'architecture pour intégrer le moteur dans l'univers. Nous allons faire en sorte que le moteur ait une influence sur les particules issues du même univers.

3.1 Intégration de MoteurCC dans l'architecture "Univers"

dans la classe 'Univers', nous avons rajouter un attribut **moteur** qui pourra être initialisé avec l'un des paramètres du constructeur (par défaut, nous avons mis 'moteur=ControlPID_vitesse' dans le code). La méthode **simulate** a été modifié afin de lancer la simulation du moteur et en précisant une vitesse cible. A chaque pas de temps, nous allons extraire la valeur du couple moteur. Puis, pour chaque individu présent dans la population de l'univers, nous calculons la distance d entre cet individu et le centre du moteur, que nous convertissons ensuite en type 'Vecteur3D'. A partir ce cela, on calcule la force que va générer le moteur sur l'individu et on le rajoute dans l'attribut **forces** de l'individu. On utilise la formule:
$F = \frac{\Gamma}{d}$ avec F : la force du moteur(N), Γ : le couple moteur(Nm) et d : distance moteur-individu (m).
ps : dans le code, nous avons rajouté un facteur 1000 dans le calcul de la force du moteur pour mieux voir l'influence du moteur dans la simulation

```python
def simulate(self):
    #Simulation du moteur
    self.moteur.simule(self.step)

    #Vitesse cible du moteur (par défaut, la vitesse augmente d'1 rad/s à chaque seconde,
    #à modifier si on veut imposer une vitesse particulière)
    self.moteur.setTarget(self.now)

    #Couple moteur
    Cm = self.moteur.moteur.getTorque()

    for agent in self.population:
        #Calcul de la distance entre l'agent et le centre du moteur
        vec_dir = agent.getPosition() - self.moteur.moteur.center
        distance = vec_dir.mod()

        #On enregistre la vitesse du moteur et la distance selon le nom de la particle qui nous intéresse
        if agent.name == "Particule 1":
            self.distances.append(distance)
            self.motor_speeds.append(self.moteur.moteur.getSpeed())

        if distance != 0:
            #Calcul de la force du moteur sur l'agent
            direction = vec_dir.norm()
            moteur_force = direction * (Cm / distance) * 1000
            #Rajout de la force du moteur dans les forces de l'agent
            agent.forces += moteur_force

        agent.pfd(self.step)

    self.time.append(self.time[-1] +  self.step)
```

Figure 5: Methode **simulate** de 'Univers' avec l'intégration du moteur

3.2 Système moteur + ressort + particule

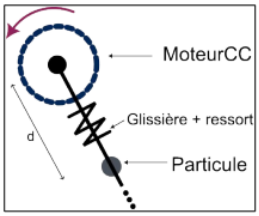


Figure 6: Système moteur + ressort + particule

Nous allons essayer de simuler le système de la figure ci-dessus. Nous voulons que la vitesse de rotation du moteur influence sur la distance d. Voici les étapes pour faire la simulation:

- Création du moteur et du contrôleur : on défini un moteur 'MoteurCC' en prenant en paramètre les mêmes valeurs que dans la partie 2 pour les grandeurs physiques, et on défini aussi le centre du moteur dans la simulation. On défini un contrôleur 'ControlPID_vitesse' pour avoir un système en boucle fermée (il n'est pas intéressant d'étudier le cas d'une boucle ouverte car pour des valeurs fixes des grandeurs physiques, la vitesse reste inchangée) qui prend en paramètre le moteur et on ajuste correctement les gains pour avoir une réponse indicielle stable et sans oscillations.

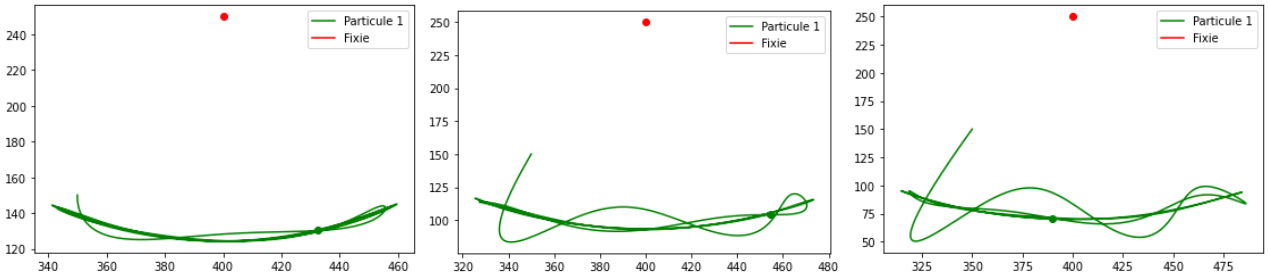
- Création des particules : on crée une particule fixe qui a la même position que le centre du moteur et une autre particule dont on veut étudier la distance au centre du moteur.

- Création des forces : Les forces qui vont nous intéresser seront la gravité et une force prismatique. Pour cette dernière, on a défini k=1 et c=0.3 pour faire en sorte que la force

de ressort n'est pas beaucoup d'influence sur la glissière. Il est nécessaire de rajouter un amortisseur car sinon la force du ressort va augmenter au fil du temps et rendre la simulation de la pendule instable.

- Création de l'univers : on crée un univers qui va prendre en paramètre le contrôleur et dans lequel on va intégrer les particules et les forces. On lance la simulation en temps réelle.

(fichier de la simulation : Run_Test_ParticuleMoteur.py)



(a) Um = 1 **(b)** Um = 50 **(c)** Um = 100

Figure 7: Déplacement de la particule pour différentes vitesse de rotation

On constate grâce à la figure ci-dessus, le fait d"augmenter la vitesse de rotation du moteur va permettre d'augmenter le couple moteur, par ce fait, la distance d va augmenter.

```
# Paramètres du moteur
R = 1  # Ω
L = 0.001  # H = Ω H
kc = 0.01  # Nm/A
ke = 0.01  # Vs
J = 0.01  # kg.m^2
f = 0.1  # Nm.s

# Création du moteur et du contrôleur
center=Vecteur3D(400, 250, 0)
moteur_bf = MoteurCC(R, L, kc, ke, J, f, center=center)
control_PID = ControlPID_vitesse(moteur_bf, Kp=5, Ki=15, Kd=0.1)

# Création des particules
P1 = Particule(mass=1, p0=Vecteur3D(350, 150, 0), v0=Vecteur3D(0, 0, 0), name="Particule 1", color="green")
P_fix = Particule(p0=center, fix=True, name="Fixie", color="red")

# Création de l'univers
espace_temps = Univers(moteur=control_PID,name='Alpha Quadrant')
espace_temps.addAgent( P1, P_fix)

# Ajout des forces
f_down = Gravity(Vecteur3D(0, -10, 0), active=True)
prismatic = Prismatic(P_fix, P1, axis=Vecteur3D(0, 0, 1))
espace_temps.addGenerators(f_down, prismatic)

# Simulation en temps réel
espace_temps.simulateRealTime(scale=1, background=(30, 30, 30))

# Affichage des résultats
espace_temps.plot()
espace_temps.plot_distance_vs_motor_speed()
```

Figure 8: Code de test du système

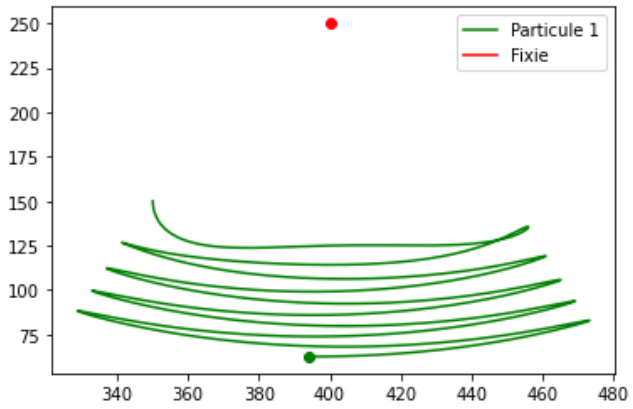
3.3 Distance d en fonction de la vitesse de rotation

Pour tracer la distance d en fonction de la vitesse de rotation du moteur, on va enregister dans des tableaux la vitesse du moteur et la distance d à chaque pas de temps durant la simulation (l'enregistrement se fait dans la méthode **simulate**). On a rajouté une méthode **plot_distance_vs_motor_speed** dans la classe 'Univers' qui va permettre d'afficher cette fonction.

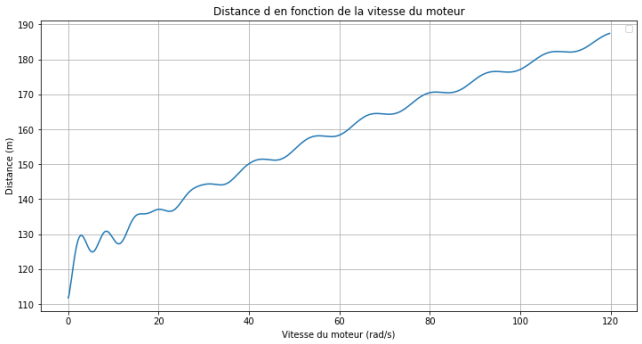
```python
#Méthode qui va afficher la distance entre le moteur et une particule en fonction de la vitesse de rotation
def plot_distance_vs_motor_speed(self):
    import matplotlib.pyplot as plt

    plt.figure(figsize=(12, 6))
    plt.plot(self.motor_speeds, self.distances)
    plt.xlabel('Vitesse du moteur (rad/s)')
    plt.ylabel('Distance (m)')
    plt.title('Distance d en fonction de la vitesse du moteur')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Figure 9: Code pour l'affichage de la distance d en fonction de la vitesse de rotation

Pour la simulation, nous avons augmenté la vitesse de rotation de 1 rad/s à chaque seconde, pendant une durée de 2 minutes.



(a) Trajectoire de la particule



(b) distance d en fonction de la vitesse de rotation

Figure 10: Résultats de la simulation

4 Validations divers

Dans cette partie, nous allons voir divers système. Pour la validation, nous allons reprendre les codes faits durant les séances sans l'intégration du moteur.

4.1 système masse + (ressort+amortisseur) + force constante

Nous allons étudier un système masse + (ressort+amortisseur) avec application au clavier d'une force constante. Pour notre système, nous aurons besoin :

- d'une particule fixe qui sera lié avec une particule 'masse' grâce à la classe 'SpringDamper'. Nous avons proposé des paramètres de $k = 1$ et $c = 0.1$ pour avoir un système sous-amorti.
- de la force de gravité, ainsi que d'une force constante qui sera émise sur la masse lors de l'appui d'une flèche sur le clavier.

Nous avons écrit une fonction **systeme_theorique_3_1** qui va permettre d'afficher la réponse théorique du système (sans l'application de la force constante). La réponse de ce type de système est de la forme :

$$x(t) = \exp(-\zeta\omega_0 t)\left(A\cos(\omega_d t) + B\sin(\omega_d t)\right) \tag{6}$$

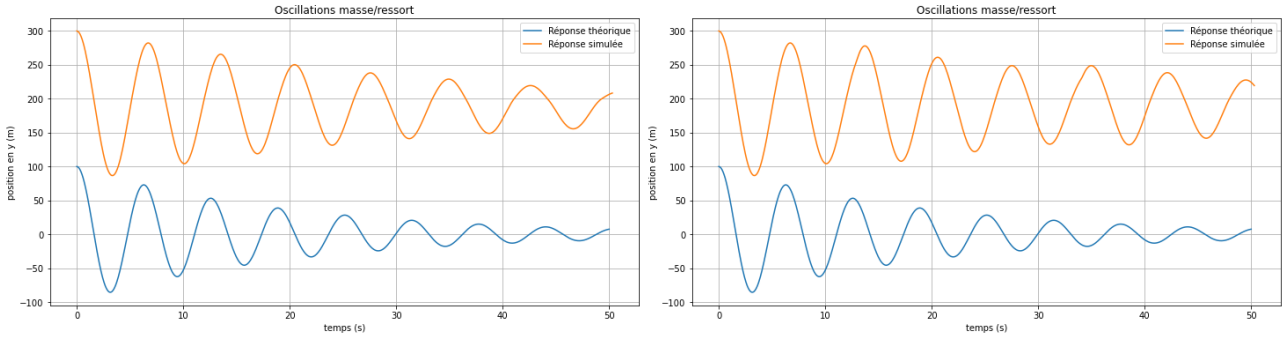
avec $\zeta = \frac{c}{2\sqrt{mk}}$: le facteur d'amortissement; $\omega_0 = \sqrt{\frac{k}{m}}$: fréquence propre non amortie
$\omega_d = \omega_0\sqrt{1-\zeta^2}$: fréquence propre d'un système sous-amorti
l'expression de A et B se trouve avec les conditions initiales.

Pour emettre la force constante avec les flèches du clavier, nous avons complété la méthode **gameInteraction** de la classe 'Univers'.

```
if self.gameMouseClick[0]:
    self.targetX = self.gameMousePos[0] / self.scale
    self.targetY = (self.H - self.gameMousePos[1]) / self.scale

for generator in self.mouseControlled:
    pass

for generator in self.keyControlled:
    generator.active = False
    if self.gameKeys[ord(' ')] or self.gameKeys[pygame.K_SPACE]: # And if the key is K_DOWN:
        generator.active =not(generator.active)

    if self.gameKeys[ord('z')] or self.gameKeys[pygame.K_UP]: # And if the key is K_DOWN:
        for agent in self.population:
            agent.forces += Vecteur3D(0, 20, 0)
    if self.gameKeys[ord('s')] or self.gameKeys[pygame.K_DOWN]: # And if the key is K_DOWN:
        for agent in self.population:
            agent.forces += Vecteur3D(0, -20, 0)
    if self.gameKeys[ord('q')] or self.gameKeys[pygame.K_LEFT]: # And if the key is K_DOWN:
        for agent in self.population:
            agent.forces += Vecteur3D(-20, 0, 0)
    if self.gameKeys[ord('d')] or self.gameKeys[pygame.K_RIGHT]: # And if the key is K_DOWN:
        for agent in self.population:
            agent.forces += Vecteur3D(20, 0, 0)
```

Figure 11: Code **gameInteraction** complété

Dans ce code, nous faisons en sorte d'appliquer une force d'unité 20 sur les particules présent dans l'Univers. La direction de la force va dépendre de la flèche du clavier sur lequel on appui.



(a) Sans application de la force constante **(b)** Avec application de la force constante

Figure 12: Réponses théorique et simulée du système

Pour la simulation, la particule a été décalé sur l'axe y afin d'avoir une meilleur visualisation durant l'affichage. Nous constatons sur la figure ci-dessus que la réponse simuléé et théorique semble tout de même différente. En effet, si on observe la figure 12 (a), il semble que la pulsation propre de la simulation soit un peu plus grande ($\omega_d \approx 6.27$ pour le calcul théorique), aussi sur la simulation, le signal semble moins atténué. Sur la figure 12(b), on peut remarquer les instants où la force est émise à l'aide du clavier avec l'augmentation de l'amplitude du signal.
(fichier pour faire la simulation : partie 3/Run_partie_3_1.py)

4.2 3 pendules

On a simulé le système de 3 pendules avec l=10, 20 et 30 cm. On utilise la classe 'Spring-Damper', on impose $k = 1$ et $c = 0.1$ pour chaque pendule.

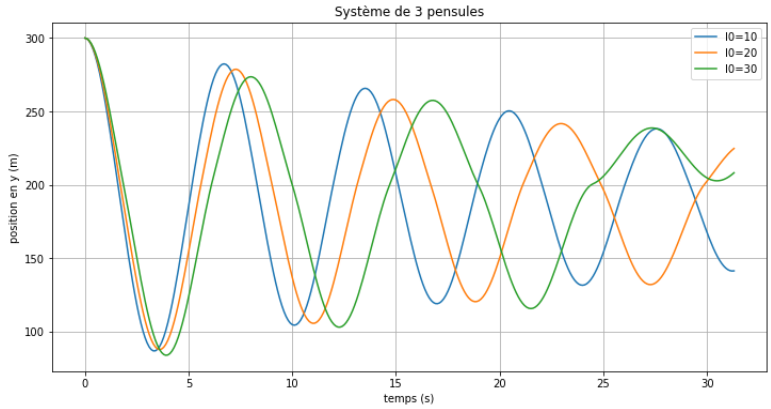


Figure 13: Position en y des 3 pendules

On constate sur la figure ci-dessus, plus la longueur l0 est grande, plus le signal diminue rapidement. En effet, cela semble cohérent car le mouvement en y est influencé plus fortement par la force de gravité, car le bras de levier est plus grand, entraînant une accélération plus rapide vers la position d'équilibre. Aussi, pour un système de pendule avec ressort, la force de rappel due à la gravité et au ressort est plus importante sur un pendule long.
(fichier pour faire la simulation : partie 3/Run_partie_3_2.py)

5 Auto-évaluation

Mon travail personnel a été marqué par une volonté de comprendre en profondeur les sujets abordés. J'ai parfois rencontré des difficultés à comprendre certains aspects abordés durant les séances, non sur l'aspect mécanique mais plus sur la manière de coder, mais ma persévérance m'a permi de surmonter ces difficultés. J'ai éssayé au mieux de suivre le travail durant les séances afin d'avoir une bonne base pour accomplir ce projet. Pour ce projet, j'ai éssayé de fournir un travail qui soit le plus compréhensible possible, mais aussi, j'ai voulu que les solutions que j'ai apporté à chaque problématique paraîsse cohérentes, notamment sur l'aspect mécanique. Je pense notamment à l'exercice 2, je me suis beaucoup demandé sur la façon dont le système est supposé marché, j'ai parfois proposé des solutions où la simulation n'était pas pertinente, ou alors la logique sur l'aspect mécanique ne me convenait pas.
La note que je m'attribue : 3/5