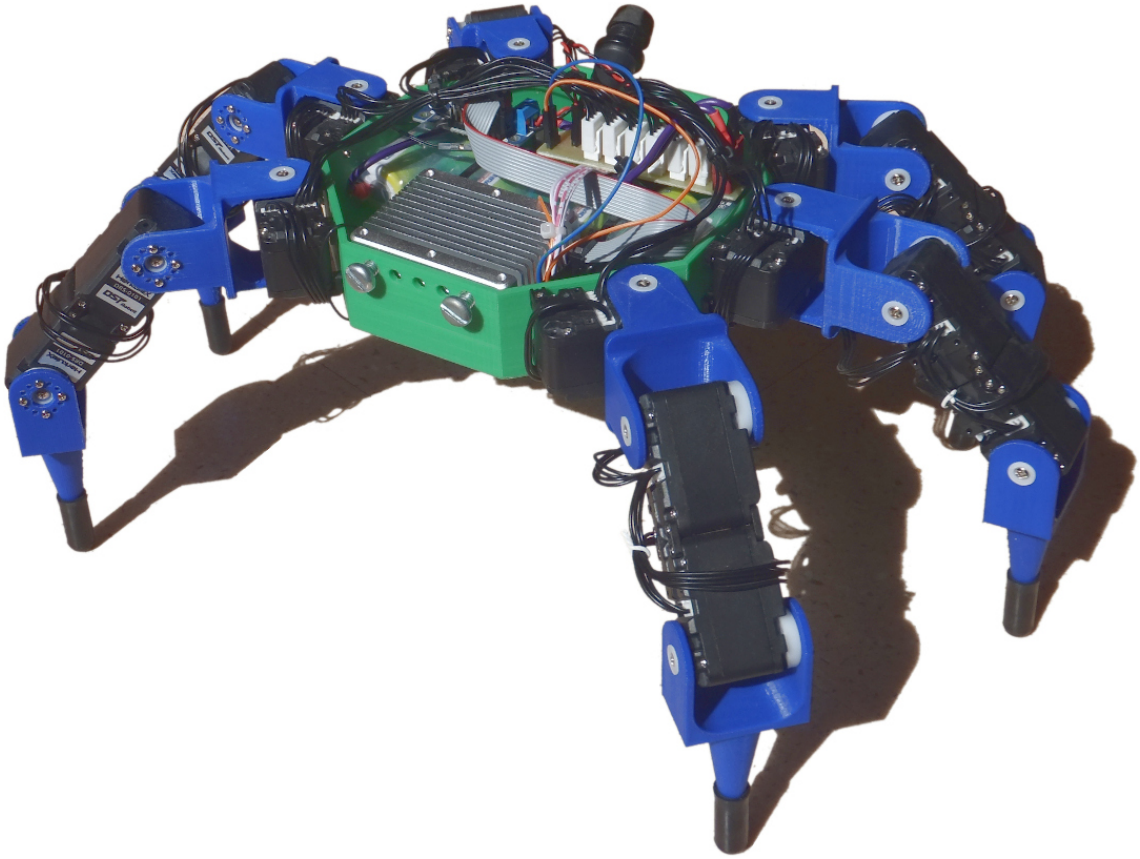
Réalisation d'un hexapode et apprentissage par renforcement de sa locomotion : de la simulation à la réalité

Travail Encadré de Recherche long à l'ENS Paris-Saclay

Alexandre IOOSS, Solal NATHAN

2019 – 2020



Résumé

Dans le cadre de notre travail encadré de recherche, nous avons travaillé sur l'optimisation de la commande d'un robot hexapode par des techniques d'apprentissage par renforcement. Notre but est d'obtenir un mouvement optimisant sa consommation énergétique et sa vitesse de déplacement tout en restant robuste. Pour cela nous avons réalisé un robot hexapode puis construit sa simulation. Nous avons également mis en place des entraînements sur l'environnement Ant d'OpenAI en utilisant le simulateur physique BulletPhysics. Le robot et la simulation obtenus constituent une base solide pour de futurs projets.

Table des matières

1 Présentation du sujet et problématique **4**

2 État de l'art **6**
2.1 Apprentissage par renforcement 6
2.2 Locomotion robotique 7

3 Composants et technologies utilisées **9**
3.1 Choix des composants pour réaliser le robot 9
3.1.1 Choix des moteurs 9
3.1.2 Choix de l'alimentation 10
3.1.3 Choix de l'ordinateur embarqué 11
3.1.4 Choix de la centrale inertielle 12
3.2 Environnement de simulation 13
3.3 Algorithmes et bibliothèques d'apprentissage 14

4 Travail réalisé **15**
4.1 Réalisation d'un robot hexapode 15
4.1.1 Modélisation et impression des pièces 15
4.1.2 Conception d'un circuit imprimé de distribution de puissance 16
4.1.3 Conception d'une carte d'extension pour le NanoPi Neo4 16
4.1.4 Mise en place de l'environnement Linux 18
4.2 Conception d'une simulation 20
4.2.1 Définition de la structure du robot 20
4.2.2 Mise en place dans PyBullet 20
4.3 Apprentissage 22
4.3.1 FrozenLake 22
4.3.2 Bandits manchots 23
4.4 Entraînement sur l'environnement Ant 26
4.5 PPO 27
4.6 Paramètres de la simulation 27

5 Conclusion **28**

Références **29**

1 Présentation du sujet et problématique

Plusieurs méthodes se sont développées pour résoudre le problème de la locomotion robotique. Il est parfois nécessaire d'utiliser un robot dans un environnement à la géométrie complexe où des roues ne peuvent pas naviguer. C'est par exemple le cas dans des zones forestières, rocheuses ou de débris après un accident. Certaines situations sont trop dangereuses pour risquer des vies humaines alors qu'il faut agir rapidement pour en sauver d'autres. Le contrôle téléguidé d'un robot n'est pas toujours possible ou précis, il faut alors permettre l'automatisation de ces techniques.

Fig. 1 : Robot hexapode tentant de rejoindre la formation SAPHIRE

Des solutions basées sur le contrôle optimal dominent [1][2], mais l'émergence de l'apprentissage par renforcement (*RL*) dans ce domaine est encourageant. Des résultats émergent des méthodes d'apprentissage dans un environnement continu comme PPO [3] [4], des méthodes d'apprentissage par renforcement inverse (*IRL*) [5] ou encore du *few shots learning*[1] pour s'adapter, par exemple, à la perte d'un membre.

Bien que les méthodes d'apprentissage puissent donner des résultats satisfaisant en simulation, il est bien plus difficile d'effectuer le passage de la simulation à la réalité [6]. En effet, des

[1] L'apprentissage en quelques coups consiste à réapprendre par transfert dans un environnement légèrement différent depuis un modèle existant.

sur-simplifications du modèle physique sont utiles pour rendre la simulation plus simple et rapide mais peuvent mener à un mauvais apprentissage. Il est difficile d'obtenir des valeurs précises de certains paramètre physiques, il va donc falloir rendre notre algorithme robuste à des variations sur ces paramètres. Enfin, l'apprentissage directement sur le robot serait beaucoup plus long qu'une simulation parallélisable.

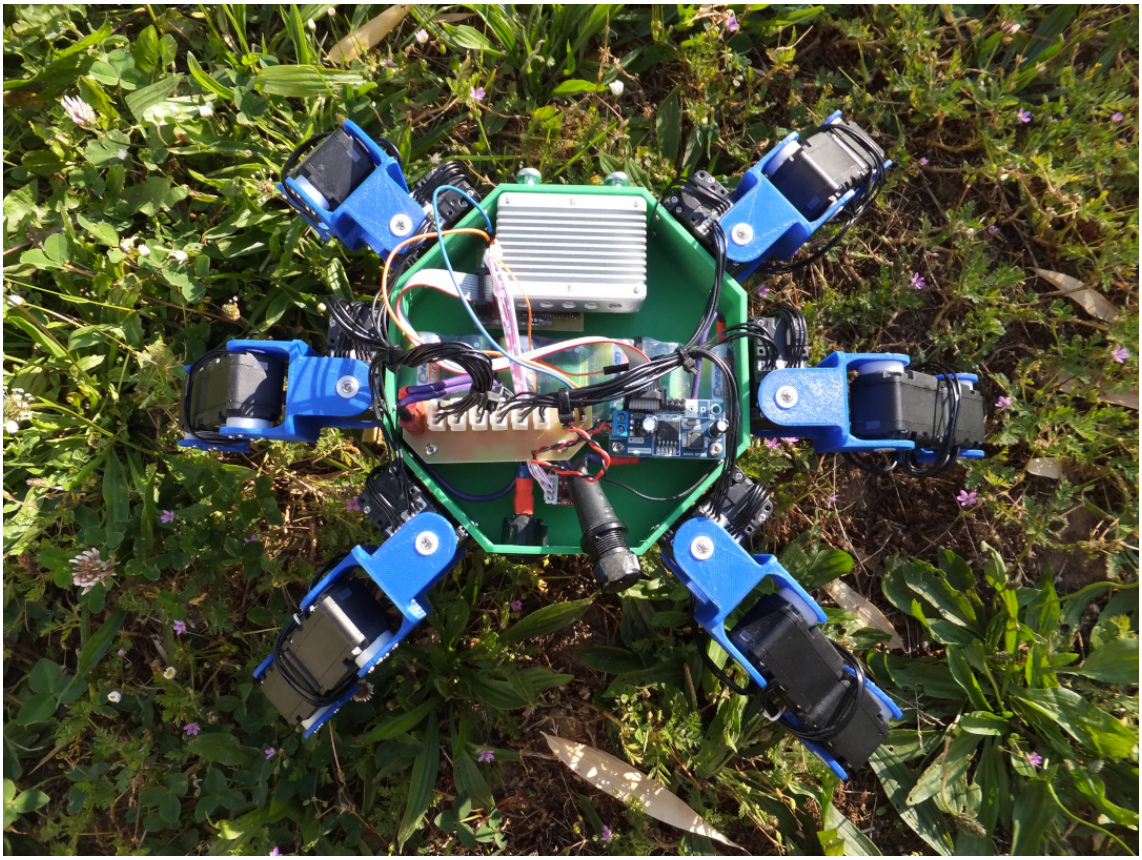


Fig. 2 : Vue de dessus du robot

Enfin, bien que les outils informatiques utilisés soient libres et open-sources, et donc facilement utilisables dans le monde de la recherche, ce n'est pas forcément le cas pour les modèles de robots que nous pouvons trouver sur le marché. Le but est donc de créer un hexapode open-hardware facilement réalisable à faible coût et modifiable selon ses besoins. Nous avons donc publié l'intégralité des fichiers permettant sa construction et sa simulation sous une licence libre.

2 État de l'art

2.1 Apprentissage par renforcement

L'apprentissage par renforcement (ou *Reinforcement Learning* en anglais, abrégé *RL*) est un domaine de l'apprentissage automatique qui consiste à rechercher une politique[2] optimale à suivre pour un agent en fonction de son environnement. Un agent est l'entité qui prend le choix d'effectuer une action a parmi l'espace d'action A, agissant ainsi dans l'environnement. En fonction de l'action choisit et de son environnement, l'agent se voit remettre une récompense r_t et un nouvel état dans lequel il se trouve à l'étape suivante [7].

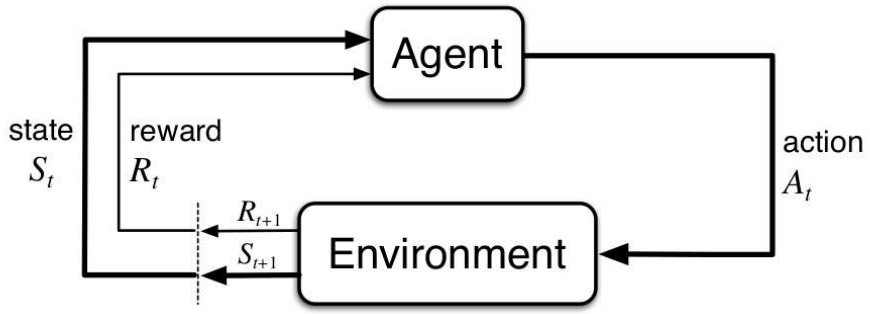


Fig. 3 : Interaction agent-environnement dans un processus de décision markovien, [7]

Ce problème est modélisé mathématiquement par des processus de décision markovien (MDP). Une méthode intuitive de résolution est d'apprendre une politique optimale au fur à et mesure des nombreuses tentatives et de les stocker dans une Q-table. Le but est donc d'apprendre la politique qui va maximiser la récompense R. En pratique, la malédiction de la grande dimension nous empêche d'utiliser cette méthode. Le Q-learning, et plus souvent le Deep Q-learning (DQN), est la méthode qui consiste à remplacer la Q-table par un réseau neuronal dont nous pouvons choisir la structure. Une preuve de la convergence du Q-learning vers une politique optimale peut être démontrée [8].

Pour résoudre le problème mathématique, nous utilisons l'équation de Bellman [9][10]. Cela permet de tendre vers une fonction Q ayant les propriétés recherchées :

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{ancienne valeur}} + \alpha \cdot \Bigg(\underbrace{r_t + \gamma \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{valeur future optimal estimée}}}_{\text{nouvelle valeur (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{ancienne valeur}} \Bigg)$$

[2]Une politique est un ensemble de principes qui permet de se guider vers des actions rationnelles. Mathématiquement elle est définie comme une fonction donnant la probabilité de choisir une action a dans un état s. Soit

$$\pi(a, s) : \begin{cases} A \times S \to [0, 1] \\ (a, s) \mapsto P(a_t = a | s_t = s) \end{cases}$$

- γ le facteur de réduction (*discount factor*)
- α le taux d'apprentissage (*learning rate*)
- r_t la récompense (*reward*)

Il existe de nombreux algorithmes de RL, celui que nous avons décidé d'utiliser est PPO [3] pour ses performances élevées sur les applications courantes de ce domaine, notamment les différents environnements de *OpenAI Gym* [11].

2.2 Locomotion robotique

La locomotion robotique reste encore aujourd'hui un problème difficile en apprentissage par renforcement. De nombreuses techniques tentent de le résoudre. Par exemple, l'apprentissage par renforcement hiérarchique (*Hierarchical Reinforcement Learning*) consiste à découper le problème en sous-problèmes et décomposer les primitives de marche en mouvements cycliques [12].

Parfois, l'environnement dans lequel le robot évolue est étroit. Il existe alors des méthodes de *bounding box* déformable qui permette au robot, ici un hexapode, de changer sa silhouette globale pour pouvoir passer dans des espaces confinés [13].

Réduire l'écart (le *gap*) qu'il existe entre la simulation et la réalité est difficile. Il existe de nombreuses techniques pour rendre l'apprentissage robuste face à ce problème [6]. Une solution est d'ajouter une certaine variabilité autour des constantes physiques mesurées (randomisation dynamique). Par exemple, la masse peut être choisie aléatoirement au début de chaque épisode de la simulation dans une distribution gaussienne centrée sur la masse mesurée. Il ne faut pas non plus que l'écart-type soit trop grand sinon l'algorithme ne sera pas capable d'apprendre correctement. Être trop robuste, ce n'est pas être généraliste mais plutôt ne rien savoir faire.

L'ajout de perturbations extérieurs, sous forme de forces placées aléatoirement, aide également à rendre la politique plus robuste aux aléas du terrains.

Enfin un espace d'observation plus compacte aide à avoir une politique optimale plus facile à calculer. Contrairement à ce que l'on pourrait croire, ce n'est pas toujours profitable de rajouter des capteurs. Un espace d'observation peu compact peut mener à un sur-apprentissage des données (*overfit*).

Si la fonction de récompense comporte un terme de pénalisation sur la consommation d'énergie, alors une démarche naturelle de galop apparaît. Pour exemple, voici la fonction de récompense issue de l'article Sim2Real :

$$r = \underbrace{(p_n - p_{n-1})d}_{\text{progrès en terme de distance}} - w \underbrace{\Delta t|\tau_n \dot{q}_n|}_{\text{terme énergétique}}$$

- p la position,
- d la direction cible,
- Δt le pas de temps,

- τ le couple dans les moteurs,
- $\dot{q}$ la vitesse angulaire des moteurs,
- $w = 0.008$ une pondération pour l'heuristique.

Il faut également rendre la simulation la plus réaliste que possible : avoir un bon simulateur physique, des mesures des caractéristiques physiques précises qui suivent des protocoles expérimentaux rigoureux, modéliser correctement les mécanismes comme les moteurs, le glissement sur le sol ou encore la latence réelle du système.

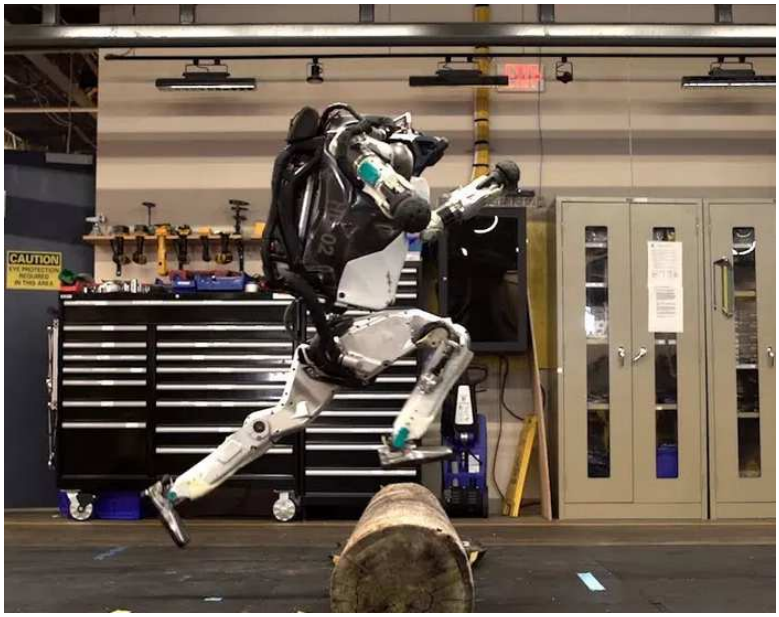


Fig. 4 : Robot Atlas en plein saut, Boston Dynamics

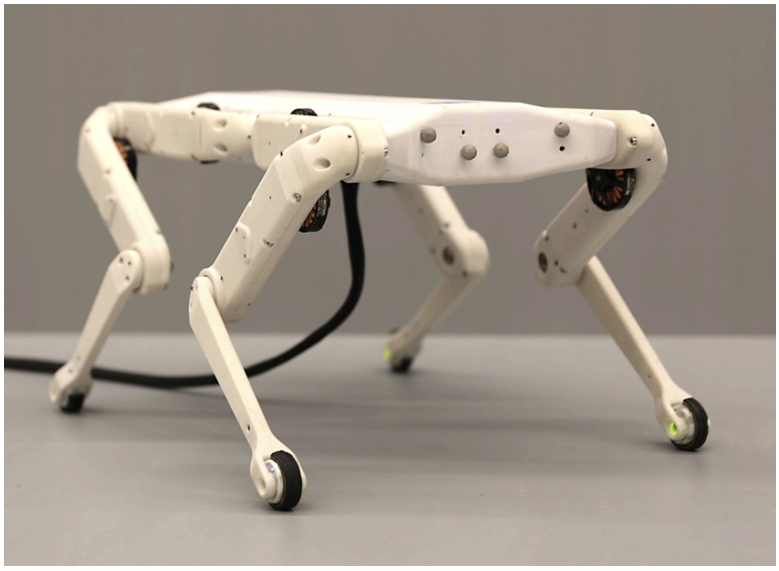


Fig. 5 : Robot SOLO, un quadrupède open-source et imprimable en 3D, Open Dynamic Robot Initiative

3 Composants et technologies utilisées

3.1 Choix des composants pour réaliser le robot

3.1.1 Choix des moteurs

Fig. 6 : `Herkulex DRS-0101` et `Dynamixel AX-12A`

Nous souhaitons utiliser des moteurs pré-asservis et commendables par une liaison série. Les `Dynamixel AX-12A` et les `Herkulex DRS-0101` correspondent aux performances nécessaires pour mouvoir notre robot, et sont communément utilisés sur des kits de robot hexapode. Ils permettent également de mesurer le couple et la vitesse en sortie du moteur ce qui permet de déduire la consommation et mesurer les performances d'une commande. Les servomoteurs Herkulex sont disponibles à un meilleur prix que les Dynamixel, au coût d'une documentation plus hasardeuse.

Le `Dynamixel AX-12A` expose une liaison série sur une seule broche, alors que le `Herkulex DRS-0101` sépare la broche d'envoi et de réception. Il est donc plus facile de commander un Herkulex.

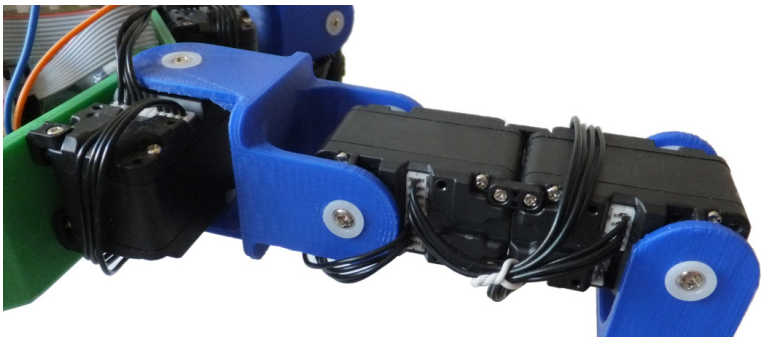


Fig. 7 : Utilisation des attaches fournies avec 2 pièces imprimées pour faire un pieds

Nous avons donc utilisé 18 servomoteurs `Herkulex DRS-0101` avec les vis et attaches fournies. Avec le recul, utiliser des moteurs avec beaucoup de frottement n'est pas idéal pour réaliser de l'apprentissage et il serait intéressant d'exploiter une alternative avec des moteurs brushless et sans étage de réduction.

3.1.2 Choix de l'alimentation

Fig. 8 : Batterie LiPo et cellule NiMH

Les moteurs choisis demande une tension continue allant de 7 à 12 V. Un moteur tire environ 500mA à 7 V en exerçant le couple maximum. L'ordinateur embarqué demande une tension continue de 5 V et peut tirer jusqu'à 3 A. Nous devons donc dimensionner une batterie capable de délivrer une tension de 7 à 12 V, à 9 A minimum puis un régulateur de tension Buck 5 V.

Pour la batterie, le choix entre une technologie à base de Lithium ou du NiMH s'est posé. Les deux technologies sont utilisées dans le modélisme et le lithium a une meilleur capacité massique, mais est plus dangereux et survit moins bien la décharge.

Par curiosité et pour tenter d'avoir une batterie qui résiste après avoir passé quelques années au fond d'une armoire d'EEA, nous avons choisi de construire une batterie de 7 cellules NiMH dans le format `Sub-C`. Nous obtenons ainsi une batterie allant de 8 à 10,5 V de 3 Ah, ce qui devrait fournir une vingtaine de minutes d'autonomie. En pratique nous nous retrouvons avec plus de 2h d'autonomie en utilisation normale.

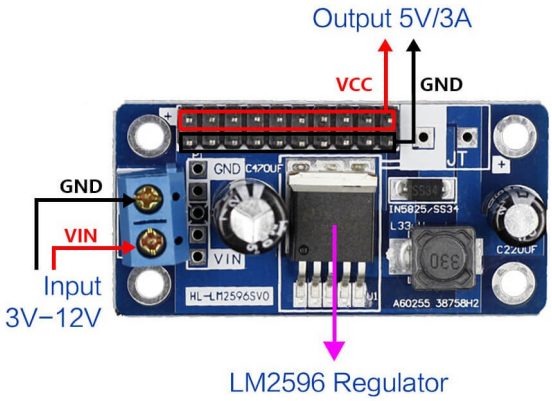


Fig. 9 : Régulateur de tension LM2596, image par Sunfounder

Il était question au début d'embarquer un régulateur de tension Buck `TI LM2596` ou `Analog ADP2303ARDZ` sur le circuit imprimé qui distribue le courant. Pour des questions de simplicité,

nous avons directement utilisé une carte contenant le `TI LM2596` de chez SunFounder. Ainsi nous avons une tension de 5 V stabilisée pour l'ordinateur embarqué.

3.1.3 Choix de l'ordinateur embarqué

Nous souhaitons embarquer un ordinateur sur le robot pour pouvoir le contrôler à distance ou lancer le réseau de neurones directement dessus en autonome. Il a donc été nécessaire de trouver une carte possédant une carte WiFi intégrée, suffisamment puissante pour exécuter le réseau de neurones, exposant un connecteur I2C et port série et le tout dans un format compact.

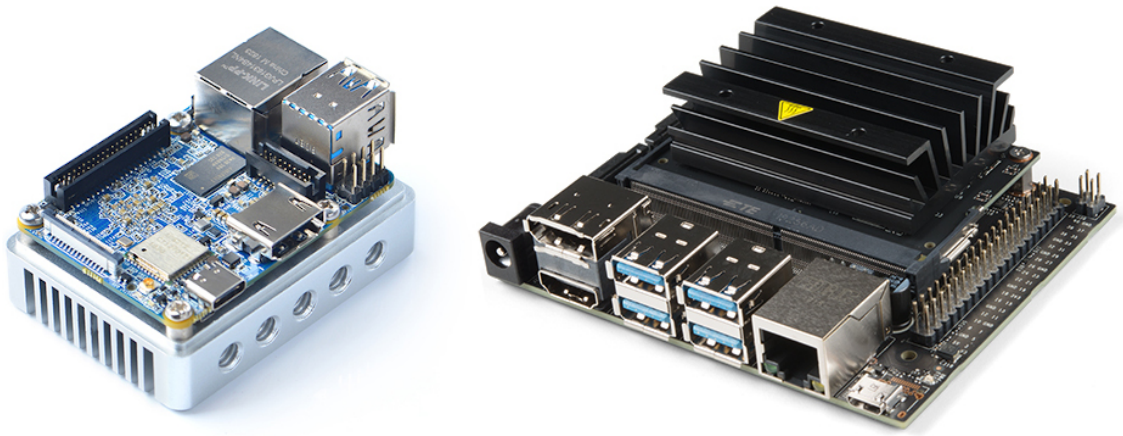


Fig. 10 : *NanoPi Neo4* et *Nvidia Jetson Nano*

Lors de l'écriture de ce compte rendu, les meilleurs rapports performance-prix tout en gardant un support officiel du noyau Linux sont les cartes équipées d'un processeur *RK3399* de Rockchip. Ce processeur ARM64 est équipé de 6 cœurs optimisés pour l'apprentissage et la vision.

Nous avons choisi d'utiliser une *NanoPi Neo4* de FriendlyElec qui se base justement sur un RK3399 dans un format deux fois plus petit qu'un Raspberry Pi B. Néanmoins la carte est utilisable seulement avec un radiateur relativement conséquent. Un défaut de cette carte est la nécessité de créer un circuit imprimé d'extension pour pouvoir exploiter la liaison série et l'I2C car le connecteur est humainement pas soudable.

Une autre option aurait été la *Nvidia Jetson Nano* qui est au double du prix de la *NanoPi Neo4*, est plus grande et a des performances comparables[3].

[3]Le RK3399 est 6% plus rapide que la Jetson Nano sur Inception-V3 et MobileNetV2 dans les exemples de Tensorflow.

3.1.4 Choix de la centrale inertielle

Le robot doit pouvoir connaître son orientation pour rester horizontal et garder sa direction. Pour cela nous utilisons une centrale inertielle équipée d'un magnétomètre.

Une centrale inertielle renvoi son orientation à partir d'un ensemble de capteurs. Des modèles comme la `MPU6050` d'InvenSense sont équipées d'un accéléromètre MEMS[4] et d'un gyromètre. Le gyromètre permet par intégration de connaître son orientation, mais dérive progressivement. Il est alors couplé à un accéléromètre qui va mesurer l'orientation de la pesanteur. L'accéléromètre est très bruité donc un filtre passe-bas sera utilisé en aval et son information permettra seulement d'éviter de dériver.

Néanmoins dans notre situation, il est souhaitable de plus que la mesure ne dérive pas selon la verticale pour maintenir le cap de l'objectif. Nous choisissons donc d'utiliser un magnétomètre pour fixer une référence magnétique et compenser la dérivation.

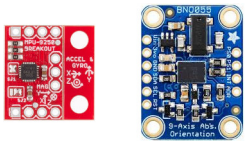


FIG. 11 : `MPU9250` d'InvenSense et `BNO055` de Bosch

La centrale inertielle `MPU9250` d'InvenSense a donc été choisie car elle est en réalité une `MPU6050` avec un magnétomètre en sous-périphérique I2C.

La `MPU6050` et la `MPU9250` sont des modèles plutôt populaires, ce qui permet d'utiliser directement le sous-système *Industrial I/O* (IIO) du noyau Linux pour la mise en place.

Il est aussi important de noter que les centrales inertielles d'InvenSense sont équipées d'un *Digital Motion Processor* qui permet de réaliser la fusion des données des deux ou trois capteurs directement sur la centrale avec un algorithme constructeur. Néanmoins nous avons choisi de plutôt récupérer les données brutes et de les traiter avec un filtre Mahony [14] ou l'amélioration de Sebastian Madgwick [15] plutôt que compter sur un algorithme fermé[5].

D'autres centrales inertielles comme la `BNO055` de Bosch propose également des fonctionnalités similaires et sont également supportées par le sous-système IIO du noyau Linux.

[4] Un microsystème électromécanique (MEMS) est un microsystème fabriqué à partir de matériaux semi-conducteurs.

[5] Le constructeur ne fournit qu'un code compilé à envoyer en I2C pour initialiser le processeur. Il n'y a pas de code source connu et en pratique le noyau Linux s'est débarrassé de cette fonctionnalité « non libre ».

3.2 Environnement de simulation

Pour réaliser la simulation d'un tel robot, il faut construire une description de sa mécanique puis utiliser un simulateur physique.

Pour la description mécanique, nous avons choisi le *Simulation Description Format* (SDFormat). Le SDFormat est une évolution du *Unified Robot Description Format* (URDF) qui ajoute les composants manquant pour spécifier l'environnement de simulation et réaliser des chaînes mécaniques fermées.

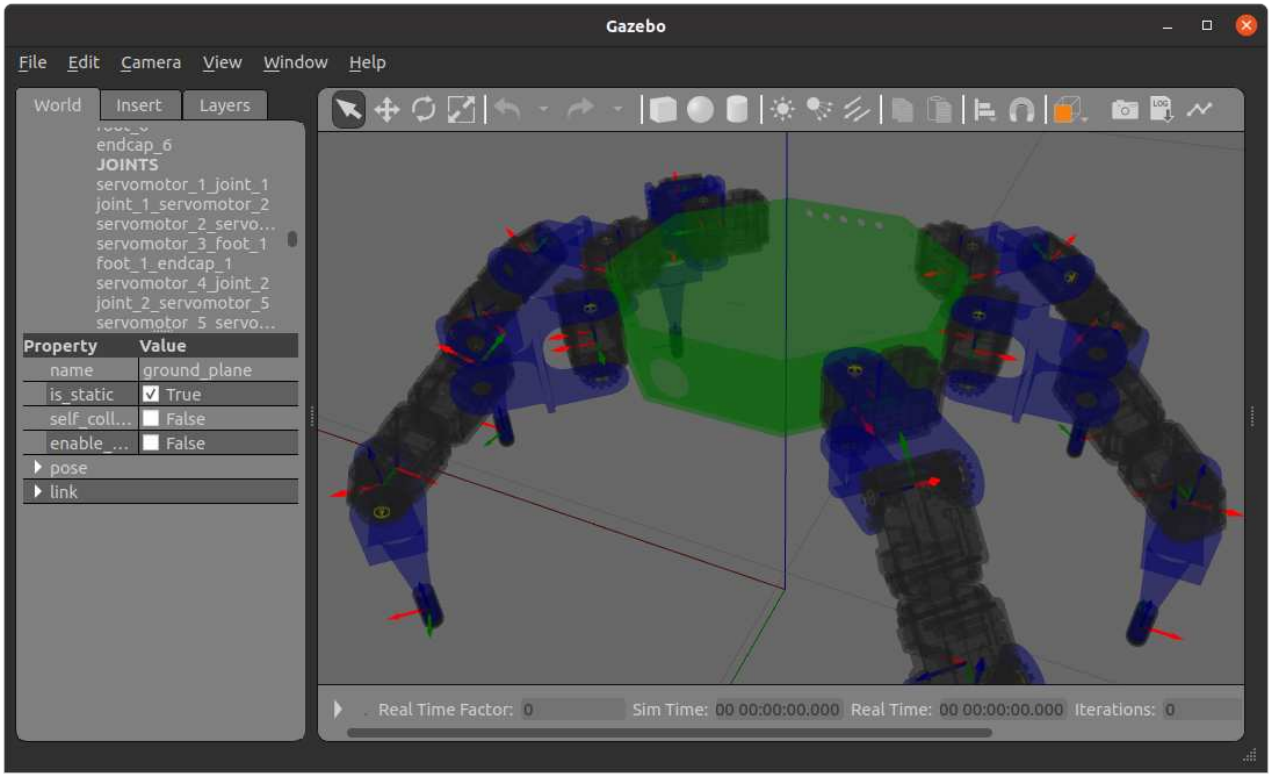


Fig. 12 : Visualisation des liaisons sur la structure de l'hexapode ouverte dans Gazebo

Le projet *Robotic Operating System* (ROS) a développé *Gazebo* qui est un outil permettant d'aider à la création et à la simulation d'un SDFormat [16]. De plus Gazebo inclut des simulateurs physiques tels que *BulletPhysics*, ODE ou *Simbody* pour directement tester son robot dans l'écosystème ROS. Le logiciel permet également de simuler des capteurs comme un lidar ou une centrale inertielle.

Néanmoins pour la simulation, nous avons choisi de directement utilisé *PyBullet* en chargeant le fichier SDFormat de notre robot. Nous avons ainsi pu construire un environnement compatible avec *OpenAI Gym*.

3.3 Algorithmes et bibliothèques d'apprentissage

Il existe de nombreuses bibliothèques qui implémentent les algorithmes de l'état de l'art en RL comme Intel Coach, Keras Rl ou OpenAI Baselines. Nous avons utilisé Baselines pour plusieurs raisons : il s'agit d'une solution libre et open-source développée par un laboratoire leader dans le domaine et qui interopère naturellement avec un ensemble d'environnements pré-existants pratiques pour apprendre et tester ses algorithmes.

Il existe notamment un environnement Ant, qui prends la forme d'une quadrupède sur lequel nous avons pu rapidement tester les différents algorithmes de l'état de l'art dans le domaine.

Fig. 13 : Ant-v2 : *Make a four-legged creature walk forward as fast as possible.*

Il s'avère qu'un *fork* de Baselines du nom de Stable Baselines [17] possède les mêmes algorithmes mais dans une structure de code unifiée, avec une documentation claire et de nombreux exemples. Le problème de la bibliothèque original de Baselines est qu'elle a été écrite dans un style peu homogène, ce qui rend le code source difficile à lire et créer de la redondance inutile dans le code. Nous avons donc choisi d'utiliser Stable Baselines pour nous fournir les algorithmes d'apprentissage par renforcement nécessaires à nos simulations.

Pour réaliser l'environnement OpenAI Gym de notre robot, nous pouvons soit utiliser Gazebo Ignition et coder les spécificités du robot en C++, soit reprendre l'environnement Ant d'OpenAI. La seconde solution a été mise en place et permet d'avoir plus de flexibilité dans le script Python pour par exemple définir le score et le comportement des moteurs.

4 Travail réalisé

Les sources des différents composants sont sur https://github.com/erdnaxe/kraby. La documentation pour réaliser le robot chez soi est sur https://kraby.readthedocs.io/.

4.1 Réalisation d'un robot hexapode

4.1.1 Modélisation et impression des pièces

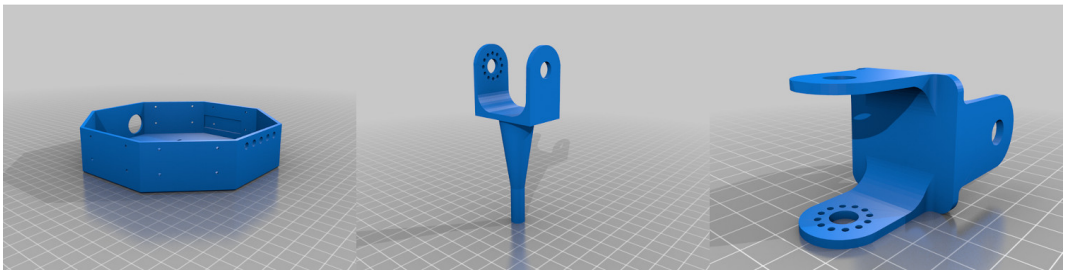


Fig. 14 : Visualisation des différentes pièces modélisées

Nous avons conçu un châssis pour réaliser un hexapode utilisant des `Herkulex DRS-0101`. Ce châssis réutilise les pièces livrées avec chaque moteur et a été pensé pour être facilement réalisable avec une imprimante 3D. La modélisation a été réalisée dans *Blender*. Nous avons imprimé les pièces avec l'ancienne Prusa i3 MK1 du Krobot, avec une hauteur de couche de 0,2 mm et la configuration par défaut du logiciel Cura.

De par la forme des liaisons des pattes et des pieds, il a fallu imprimer avec des supports pour que la gravité n'effondre pas les zones non supportées des pièces. L'impression totale (sans compter les prototypes) a représenté plus de **22h30 d'impression**, 88.28m de filament PLA consommé soit le quart d'une bobine de plastique d'un kilogramme.

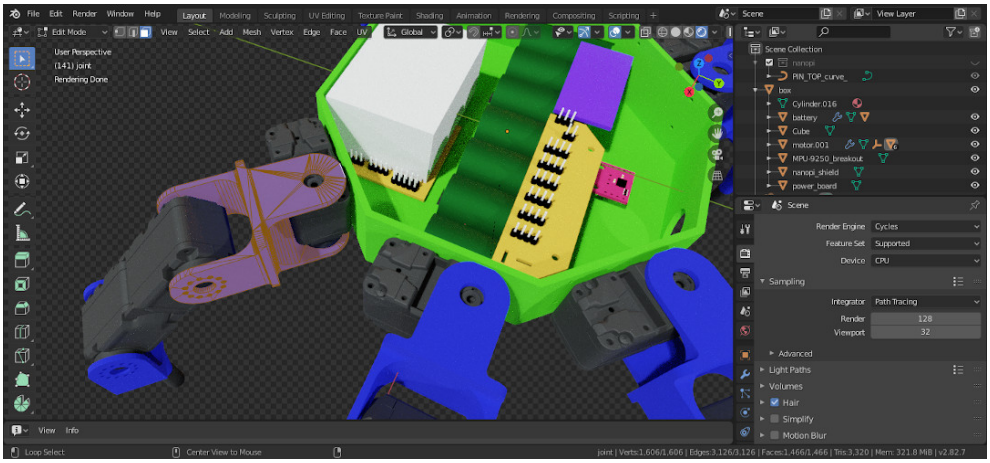


Fig. 15 : Blender en mode d'édition du nuage de points

4.1.2 Conception d'un circuit imprimé de distribution de puissance

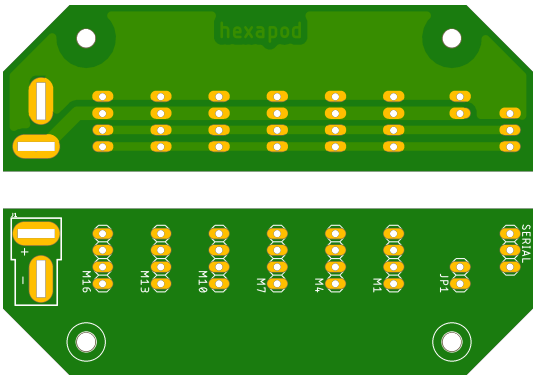


Fig. 16 : Rendu de la carte de distribution de puissance

La carte de distribution de puissance n'a aucun composant actif. Ce n'est qu'un circuit imprimé permettant de connecter plus facilement l'ensemble des moteurs à la batterie et au port série.

L'idée est de pouvoir connecter directement les Herkulex en tête des 6 pattes. Ainsi l'ensemble des servomoteurs sont en série, sans pour autant faire passer toute la puissance à travers un servomoteur, ce qui l'abîmerait.

Le TX/RX des servomoteurs sont reliés à 3 broches (TX, RX, GND) pour être connectées sur un convertisseur USB-série ou directement sur l'ordinateur embarqué.

4.1.3 Conception d'une carte d'extension pour le NanoPi Neo4

Le NanoPi Neo4 utilise un connecteur `HARWIN M50-4912045`. Nous avons donc commandé le connecteur femelle associé `HARWIN M50-4302045`. Il est ensuite question de rendre accessible le bus I2C, l'alimentation 5V et un port série avec des broches plus larges.

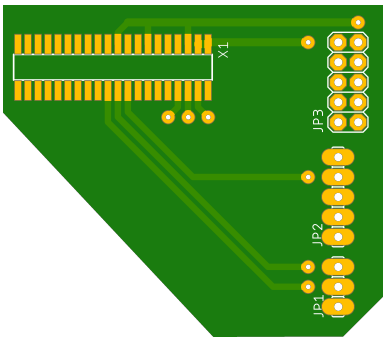


Fig. 17 : Carte d'extension pour le NanoPi Neo4

La première version de cette carte d'extension utilise les broches `I2C3_SDA/SCL` mais elles ne sont pas configurées et accessibles depuis Linux. La solution a donc été de couper la piste de

cuivre et de dévier la connexion vers `I2C2_SDA/SCL`. La broche d'interruption de la centrale inertielle était également oubliée et nécessaire pour le bon fonctionnement du module IIO. Nous l'avons donc rajouté.

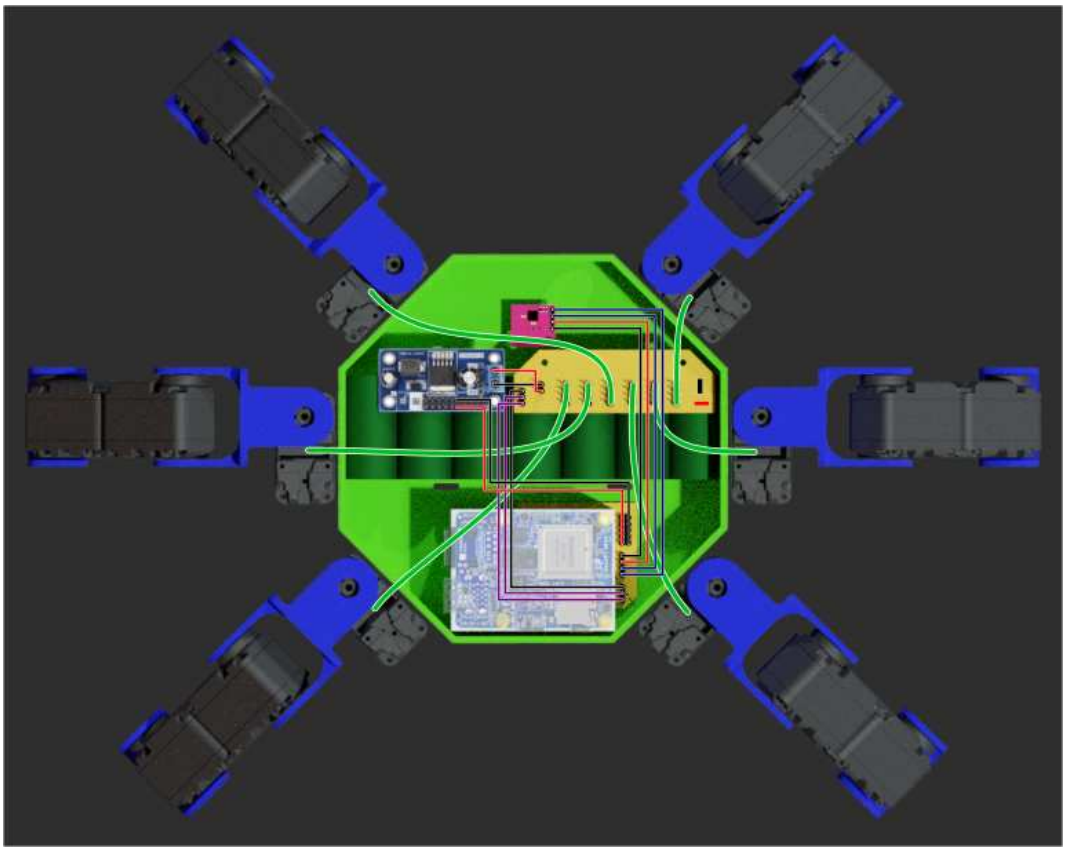


Fig. 18 : Interconnexion des différents composants

4.1.4 Mise en place de l'environnement Linux

Ce rapport n'a pas pour vocation de rentrer dans les détails pratiques qui sont disponibles sur https://kraby.readthedocs.io/en/latest/install_nanopi_linux/.

4.1.4.1 Utilisation de la centrale inertielle Un Linux embarqué sur de l'ARM64 va charger au démarrage un *device-tree* définissant l'ensemble des périphériques de la carte embarquée. Cela permet à un même noyau de supporter plusieurs matériels.

Dans notre cas nous souhaitons modifier le *device-tree* pour ajouter la centrale inertielle et recompiler Linux pour inclure le module noyau `inv-mpu6050-i2c`. Bien que le module noyau se nomme `inv-mpu6050-i2c`, il supporte entièrement la MPU9250 utilisée et il suffit de définir en sous-périphérique le magnétomètre *AK8975*. La modification du fichier source `arch/arm64/boot/dts/rockchip/rk3399-nanopi4-common.dtsi` ressemble alors à cela :

```
 &i2c2 {
     status = "okay";
+
+    mpu9250@68 {
+        compatible = "invensense,mpu9250";
+        reg = <0x68>;
+        interrupt-parent = <&gpio1>;
+        interrupts = <0 IRQ_TYPE_LEVEL_HIGH>;
+        i2c-gate {
+            #address-cells = <1>;
+            #size-cells = <0>;
+            ax8975@c {
+                compatible = "ak,ak8975";
+                reg = <0x0c>;
+            };
+        };
+    };
 };
```

L'intérêt de procéder ainsi plutôt que d'interagir avec le périphérique I2C en Python ou C est de profiter du sous-système *Industrial I/O* de Linux, qui permet entre autre de publier ces capteurs sur le réseau en installant `iiod`. Pour récupérer les données, il suffit alors de définir une fréquence d'acquisition et sélectionner les données souhaitées, puis une mémoire tampon sera constamment provisionnée.

4.1.4.2 Utilisation des moteurs Pour ce qu'est des moteurs, l'interface série est disponible sur `/dev/ttyS4`. Il suffit donc de publier ce TTY sur un socket réseau TCP avec par exemple le programme `ser2net`. Nous pouvons ensuite utiliser le module `socket` de Python pour commander les moteurs et récupérer leurs mesures.

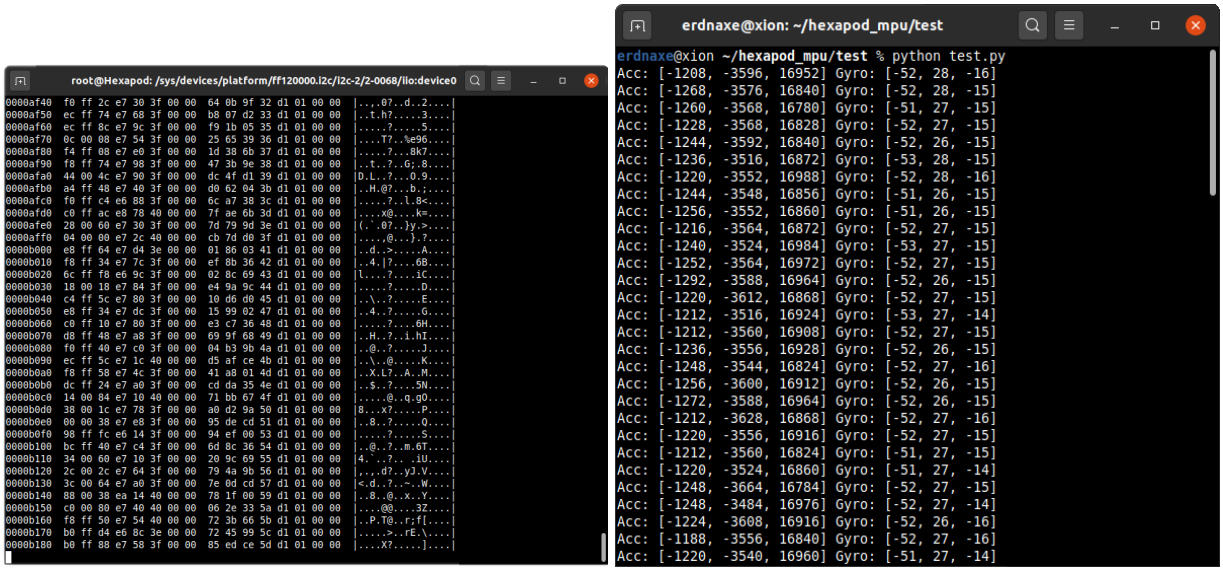


FIG. 19 : Données brutes récupérées avec un `hexdump` de `/dev/iio :device0` sur la carte embarquée puis avec un script Python utilisant le module `iio` sur un ordinateur connecté en WiFi au robot

4.2 Conception d'une simulation

Les sources de la simulation sont publiées sur https://github.com/erdnaxe/kraby_sdf.

4.2.1 Définition de la structure du robot

Il a tout d'abord été question de définir la structure du robot dans le format SDFormat. Pour cela nous avons écrit un fichier XML contenant chaque pièce statique, puis les liaisons. De plus ce format permet d'attacher des inerties et de placer des capteurs ou actionneurs.

Pour obtenir les inerties, nous avons utilisé les protocoles suivant :

- Pour une pièce imprimée en 3D, nous la supposons homogène puis par mesure de sa masse et calcul numérique avec *MeshLab* nous obtenons l'inertie.
- Pour le corps du robot et les moteurs nous les supposons homogènes. Cela est une approximation qui peut mener à une simulation non réaliste mais en pratique il semblerait que cela soit suffisant.

4.2.2 Mise en place dans PyBullet

Ensuite que la structure a été définie dans un SDFormat, nous pouvons l'importer dans *BulletPhysics* directement et s'affranchir de *Gazebo*. Le code Python pour cela est plutôt intuitif,

```
import pybullet as p
import pybullet_data
import os, time

p.connect(p.GUI)  # ouvre la fenêtre
p.setGravity(0, 0, -9.81)  # pesanteur

p.setAdditionalSearchPath(pybullet_data.getDataPath())
p.loadSDF("plane_stadium.sdf")  # on charge un sol

p.setAdditionalSearchPath(os.getcwd())
p.loadSDF("hexapod.sdf")  # on charge le robot

while True :
    p.stepSimulation()
    time.sleep(1./60.)  # 60 fps

p.disconnect()
```

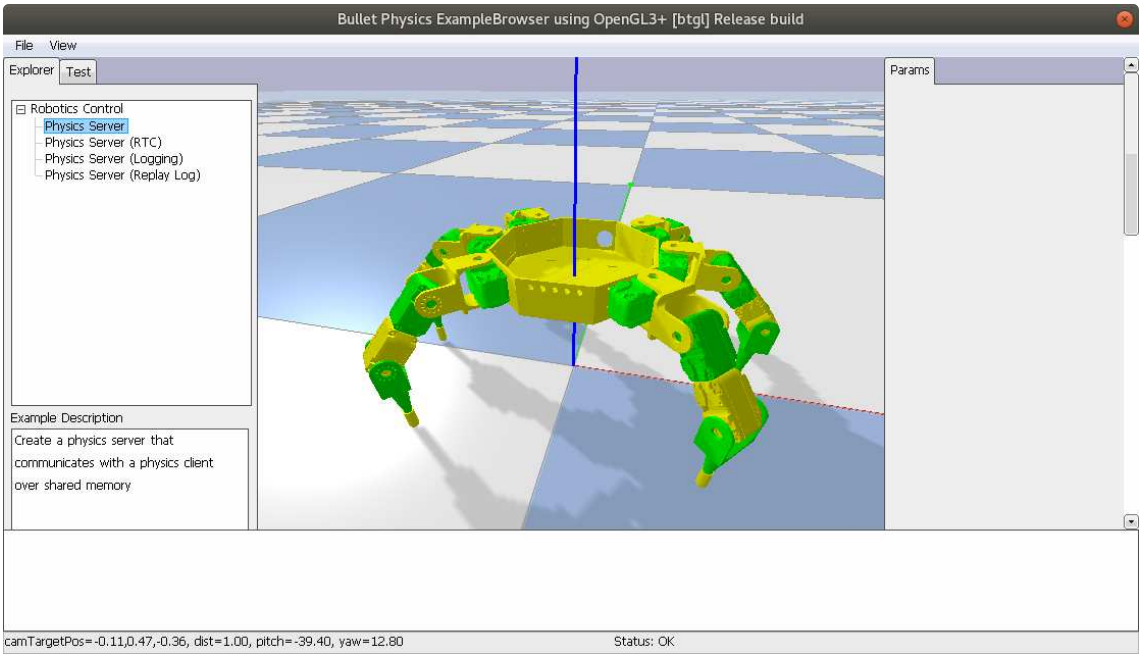


FIG. 20 : Robot chargé dans un script Python utilisant PyBullet

4.3 Apprentissage

En RL, il est assez commun de régler ses heuristiques manuellement. Il existe des méthodes pour optimiser ses hyper-paramètres[18], mais nous n'allons pas les explorer durant ce projet. Il nous faut donc un moyen de comprendre finement les différentes heuristiques auxquels nous allons faire face. Pour cela nous allons travailler avec plusieurs modèles-jouets[6] pour nous guider dans le choix de nos hyper-paramètres et mieux comprendres nos problèmes.

4.3.1 FrozenLake

Un des environnement fourni par OpenIA se nomme `FrozenLake`. Il s'agit d'un environnement très simple dans lequel l'apprentissage d'une politique stochastique est une solution évidente. Un agent déplace un personnage sur une grille. Sur certaines cases de la grille on peut se déplacer, et sur d'autres on tombe dans un trou d'eau glacée, ainsi mettant fin à l'épisode. L'agent commence tout en haut à gauche de la grille et doit atteindre un frisbee[7] en bas à droite. S'il l'atteint cela met également fin à l'épisode d'apprentissage. Malheureusement le sol est glissant et parfois une décision est prise de façon aléatoire à la droite ou la gauche de l'endroit où l'on voudrait aller.

Fig. 21 : FrozenLake-v0

[6]Modèle simplifié pour comprendre plus facilement un phénomène.
[7]disque-volant

En jouant sur la valeur du facteur d'apprentissage α nous observons le phénomène suivant :

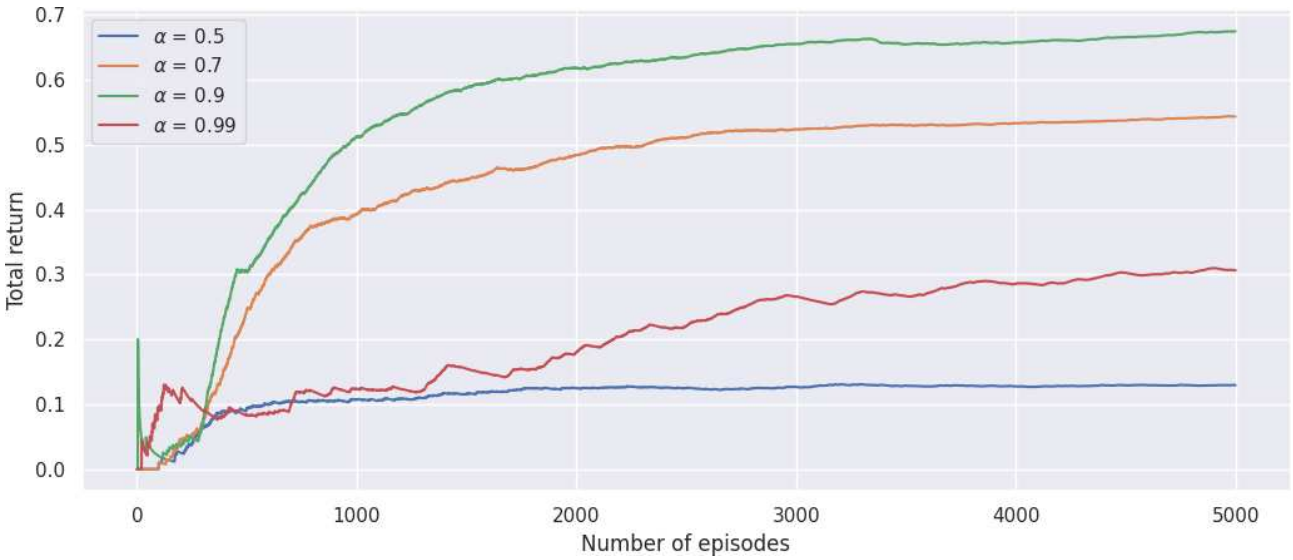


FIG. 22 : Récompense en fonction du nombre d'épisodes pour différents α

Nous observons que quand α augmente, on récupère des récompenses de plus en plus grandes. Mais lorsque que l'on dépasse un certains seuil, la récompense diminue de nouveau (voir $\alpha = 0.99$). Nous avons ici choisi une valeur de α qui est constante au long des épisodes, mais nous pourrions tout à fait la faire varier. Nous faisons face ici au dilemme exploration-exploitation.

4.3.2 Bandits manchots

L'analogie classique permettant de parler du dilemme exploration-exploitation est celle des bandits manchots. Imaginons un ensemble de machine à sous. Nous ne connaissons pas a priori si elles sont différentes les unes des autres et l'espérance de gain que l'on peut tirer de chacune d'elles. Quelle est alors une stratégie optimale pour récupérer le plus d'argent en le moins de tentatives possibles ?

On peut s'intéresser à différentes fonctions pour la suites des $\{\alpha_t\}_{t\in\mathbb{R}}$ et à une stratégie ε-gloutonne (ε-*greedy*). C'est une stratégie qui consiste à explorer ε% du temps et à exploiter $(1-\varepsilon)$% du temps.

Si on choisit par exemple d'uniquement observer dans un premier temps, puis d'uniquement exploiter dans le second temps on obtient le résultat suivant :

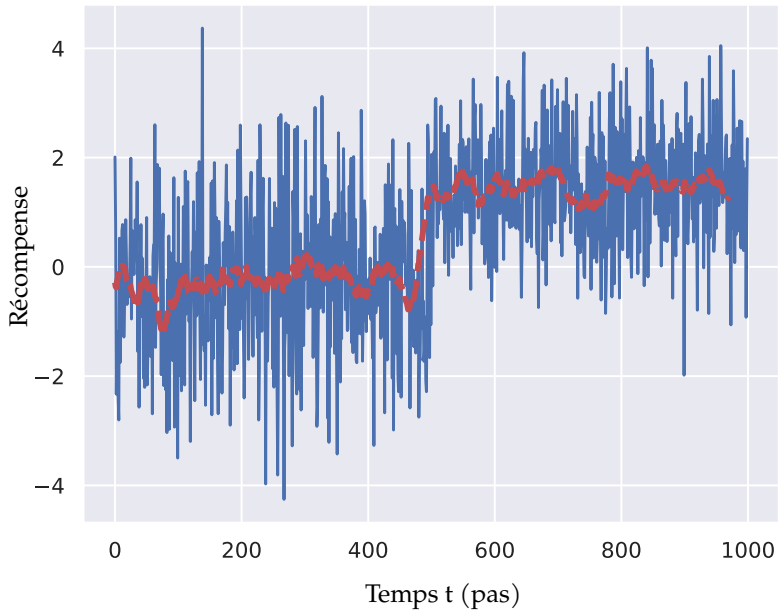


Fig. 23 : Exploration quand $t < 500$ puis exploitation

On trace la récompense en fonction du temps en bleu et la moyenne glissante de cette récompense en rouge. On obtient une récompense total moyenne d'environ 700.

Si on prend une valeur constante de $\varepsilon = 0.2$ on obtient une valeur moyenne plus haute.

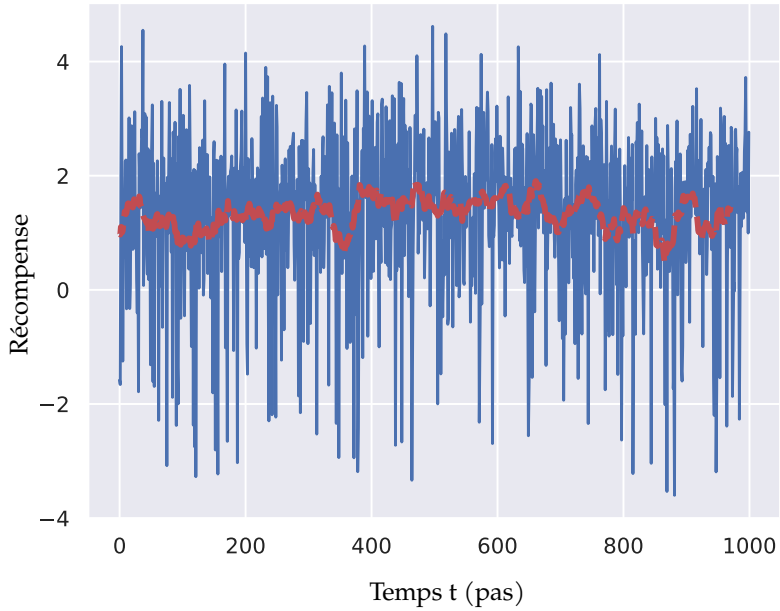


Fig. 24 : Stratégie ε-greedy pour $\varepsilon = 0.2$

On obtient une récompense total d'environ 1300.

Il est alors facile d'imaginer un grands nombre de fonctions pour $\varepsilon(t)$ qui peuvent mener à de meilleurs résultats.

Mais il est parfois possible que l'environnement soit non stationnaire. Dans ce cas là, la stratégie d'exploration puis d'exploitation ne pourrais pas fonctionner. Mais une stratégie ε-gloutonne avec un ε constant fonctionnerait.

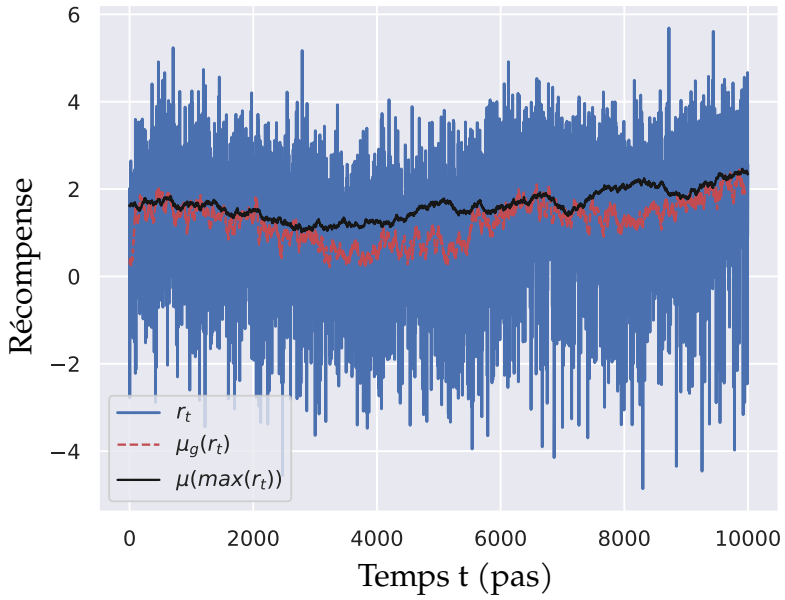


Fig. 25 : Stratégie ε-greedy pour $\varepsilon = 0.1$ pour un problème stationnaire

Ici, en plus de tracer la moyenne glissante de la récompense on trace la moyenne de la meilleur récompense possible à chaque instant. On remarque que cette stratégie permet d'atteindre des performances proche de l'optimal.

Il existe d'autres solutions à ce problème comme *Upper-Confidence-Bound Action Selection* (UCB) par exemple. Ces différentes stratégies sont résumées avec le diagramme suivant [7] :

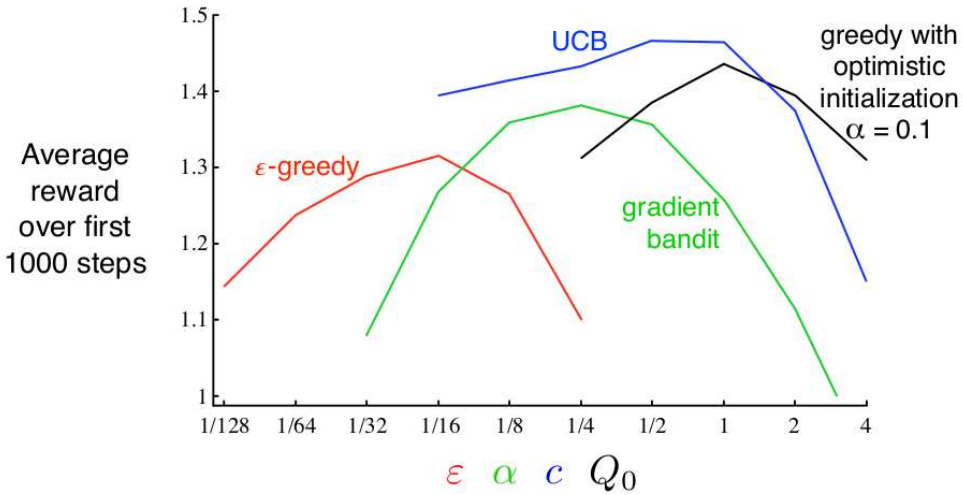


Fig. 26 : Résumé des performances de différents algorithmes de bandit, extrait de [7]

4.4 Entraînement sur l'environnement Ant

Nous aurions voulu avoir la chance de travailler sur un modèle de notre hexapode simulé, mais nous n'avons pas eu le temps de le mettre en place. Nous avons donc décidé d'entraîner sur un modèle similaire : `Ant-v2`. Il s'agit d'un robot quadrupède, qui a une structure similaire à notre hexapode.

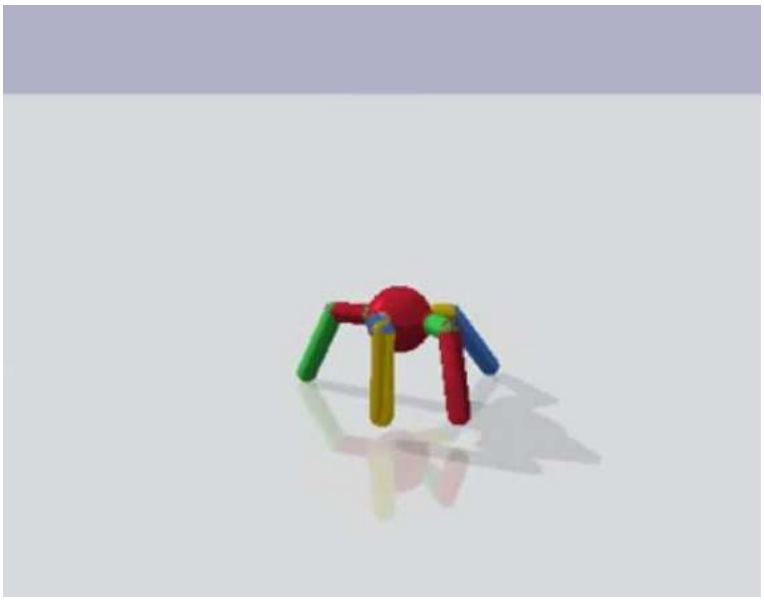


FIG. 27 : Modèle Ant à la fin de notre entraînement

Nous avons eu la chance de pouvoir récupérer les hyper-paramètres optimisés grâce à Optuna [18] pour ce modèle (`AntBulletEnv-v0` pour être précis) grâce à la bibliothèque Baselines Zoo [19] [17]. Pour des raisons de puissances de calcul à notre disposition nous avons du revoir à baisse le nombre d'épisodes totaux. Les résultats de l'entraînement avec PPO restent tout de même satisfaisant et le robot acquiert une démarche lui permettant de se déplacer sur de longue distances.

Les hyper-paramètres choisis sont les suivants :

- On choisit un réseau perceptron multicouche (Multi Layer Percetron - MLP) qui possède deux couches cachées (hidden layers) de 64 noeuds chacune.
- Un nombre total de pas de temps de 1×10^5 (valeur revu à la baisse pour des raisons de performances).
- Une taille de batch de 32.
- Une cliprange $\varepsilon = 0.2$ (valeur classique).
- Un discount factor de $\gamma = 0.99$.
- Un learning rate $\alpha = 1 \times 10^{-3}$

4.5 PPO

Nous avons choisi d'utiliser l'algorithme de PPO (Proximal Policy Optimisation) [3] pour différentes raisons. Il obtient des excellents résultats aux différents benchmarks de OpenAI Gym, il est disponible dans Baselines et utilisable dans le contexte de notre entraînement.

Le but initial de PPO est de fournir un algorithme dont la politique ne change pas trop d'un pas à un autre pour éviter de faire des pas trop grands et donc de rater une optimisation vers un minimum local. Les facteurs pris en compte par OpenAI pour développer cet algorithme sont la simplicité de son implémentation, sa *sample efficency* et sa facilité à être régler (*hyper-parameter tuning*). PPO ne s'éloigne pas trop de sa politique précédente grâce à son terme de cliping dans sa fonction de descente de gradient de la politique (*policy grandient loss*).

$$L^{CLIP}(\theta) = \hat{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t) \right]$$

- ε la cliprange,
- r_t la récompense (*reward*),
- θ les pondérations du réseau de neurones,
- $\hat{E}_t$ l'espérance empirique sur un nombre fini d'ensemble de mesures,
- $\hat{A}_t$ l'estimer de l'action à l'instant t.

Enfin, l'une des spécificités de l'algorithme PPO est qu'il s'agit d'une méthode on-line (sans replay buffer contrairement à ACER par exemple), c'est-à-dire qu'il ne peut pas apprendre d'expériences passées, mais uniquement des actions en cours.

4.6 Paramètres de la simulation

Durant la simulation du modèle de `Ant` nous avons pu explorer plusieurs points intéressants. OpenAI Gym fournit un grand nombre de primitives que l'on peut réutiliser. Par exemple une simulation de moteur et le correcteur qui l'accompagne en python.

Il faut également choisir comment mettre fin à un épisode d'apprentissage. Dans notre cas si le robot tombe ou que le nombre de pas de temps maximal est atteint, alors la simulation passe à l'épisode suivant. Nous avons enfin repris la fonction de récompense proposé le papier Sim2Real [6] évoquée dans une section précédente. Il suffit alors de fournir une direction d dans laquelle le robot doit se déplacer.

5 Conclusion

Fig. 28 : Le robot réalisé devant l'ancien bâtiment de l'ENS Paris-Saclay

Ce projet nous a permis de découvrir et mettre en place de nombreux outils pour réaliser un robot à partir de rien, concevoir sa simulation et mettre en place un environnement d'entraînement.

Bien que nous ayons découvert et appris beaucoup sur l'entraînement profond, nous n'avons pas eu le temps de le mettre en place sur ce robot en particulier. Néanmoins nous avons fait en sorte que n'importe qui soit capable de réutiliser notre conception de l'hexapode et sa simulation. Nous espérons que ce projet soit ainsi repris et amélioré.

Le robot conçu est une base de développement pensée pour être facilement réutilisée et modifiée. Bien que se structure soit réalisable à moindre coût, elle a tenu de nombreux chocs et déménagements. L'ensemble du robot revient aux alentours de 850 €, ce qui est inférieur aux kits équivalents du marché. La majorité du budget correspond aux servo-moteurs. Des laboratoires de recherche auront donc la possibilité d'économiser en le réalisant eux-même et la liberté de le modifier.

Nous n'avons pas mis en place de capteur aux extrémités des pieds du robot pour détecter le sol. Il se pourrait que ces capteurs apportent un confort lors de la mise en place de l'apprentissage, mais nous ne pouvons pas encore statuer si cela est nécessaire.

Finalement ce TER a permis de réaliser une base solide pour des futurs projets et nous espérons voir ce robot apprendre à marcher.

Références

[1] G. Wiedebach, S. Bertrand, T. Wu, L. Fiorio, S. McCrory, R. Griffin, F. Nori et J. Pratt, *Walking on Partial Footholds Including Line Contacts with the Humanoid Robot Atlas*, 2016. arXiv : 1607.08089 [cs.RO].

[2] S. Feng, E. Whitman, X. Xinjilefu et C. G. Atkeson, "Optimization based full body control for the atlas robot," in *2014 IEEE-RAS International Conference on Humanoid Robots*, 2014, p. 120-127.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford et O. Klimov, *Proximal Policy Optimization Algorithms*, 2017. arXiv : 1707.06347 [cs.LG].

[4] J. Schulman, P. Moritz, S. Levine, M. Jordan et P. Abbeel, *High-Dimensional Continuous Control Using Generalized Advantage Estimation*, 2015. arXiv : 1506.02438 [cs.LG].

[5] A. Y. Ng et S. J. Russell, "Algorithms for Inverse Reinforcement Learning," in *Proceedings of the Seventeenth International Conference on Machine Learning*, sér. ICML '00, San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000, p. 663-670, isbn : 1558607072.

[6] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez et V. Vanhoucke, *Sim-to-Real : Learning Agile Locomotion For Quadruped Robots*, 2018. arXiv : 1804.10332 [cs.RO].

[7] R. S. Sutton et A. G. Barto, *Reinforcement Learning : An Introduction*, 2nd Ed. The MIT Press, 2018.

[8] F. S. Melo, *Convergence of Q-learning, a simple proof*. Institute for Systems et Robotics, 2007.

[9] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas et D. P. Bertsekas, *Dynamic programming and optimal control*, 2. Athena scientific Belmont, MA, 1995, t. 1.

[10] S. M. Ross, *Introduction to stochastic dynamic programming*. Academic press, 2014.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang et W. Zaremba, *OpenAI Gym*, 2016. arXiv : 1606.01540 [cs.LG].

[12] T. Li, N. Lambert, R. Calandra, F. Meier et A. Rai, *Learning Generalizable Locomotion Skills with Hierarchical Reinforcement Learning*, 2019. arXiv : 1909.12324 [cs.RO].

[13] R. Buchanan, T. Bandyopadhyay, M. Bjelonic, L. Wellhausen, M. Hutter et N. Kottege, *Walking Posture Adaptation for Legged Robot Navigation in Confined Spaces*, 2019. arXiv : 1901.10863 [cs.RO].

[14] R. Mahony, T. Hamel et J. Pflimlin, "Nonlinear Complementary Filters on the Special Orthogonal Group," *IEEE Transactions on Automatic Control*, t. 53, n° 5, p. 1203-1218, 2008.

[15] S. O. H. Madgwick, A. J. L. Harrison et R. Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm," in *2011 IEEE International Conference on Rehabilitation Robotics*, 2011, p. 1-7.

[16] N. Koenig et A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, sept. 2004, p. 2149-2154.

[17] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor et Y. Wu, *Stable Baselines*, https://github.com/hill-a/stable-baselines, 2018.

[18] T. Akiba, S. Sano, T. Yanase, T. Ohta et M. Koyama, "Optuna : A Next-generation Hyperparameter Optimization Framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[19] A. Raffin et Al., *RL Baselines Zoo*, https://github.com/araffin/rl-baselines-zoo, 2018.