Équipe :
Étudiant 1 - ******* *********
Étudiant 2 - **** *********
Étudiant 3 - ******* ****
Étudiant 4 - ****** ********

LYCÉE
FÉLIX
LE DANTEC
du secondaire au supérieur

Rapport de projet individuel

Projet Robotique

Présenté par

**** *******

BTS Système Numérique option IR

Année scolaire 2023-2024

Sommaire

Annexes

Liste des figures

Introduction

Le but du projet est de déplacer une pièce, d'un point A à un point B, qui peuvent être représentés par des zones de stockage (par exemple), en utilisant deux robots qui prendront et poseront les pièces.

Le système à automatiser se compose d'un convoyeur (tapis roulant) et de deux robots. Le premier est un robot industriel **FANUC**. Ce robot est implanté dans l'atelier du lycée et est sécurisé par une enceinte grillagée (le système se trouvera donc dans cette zone).
Le second est un robot didactique[1] **Niryo Ned 2**. Le pilotage est contrôlé par un écran tactile communiquant en **Modbus/TCP**.

L'ensemble de la partie opérative (PO) est supervisé par une application graphique. La communication avec les différents éléments utilise le protocole **Modbus/TCP**.

Le cycle principal de production peut se résumer cela :

- Si une pièce est disponible, alors le robot **Fanuc** prend la pièce dans le bac et la pose sur le tapis.
- Si le capteur d'entrée du tapis est activé, alors faire avancer le tapis jusqu'à ce que le capteur de sortie soit activé.
- Robot **Niryo** prend la pièce et la dépose dans le second bac.
- Et ainsi de suite…

1 Un robot pour l'apprentissage, l'éducation

1. Situation dans le projet

Voici une représentation 3D du système / projet. (Image créé par ******)

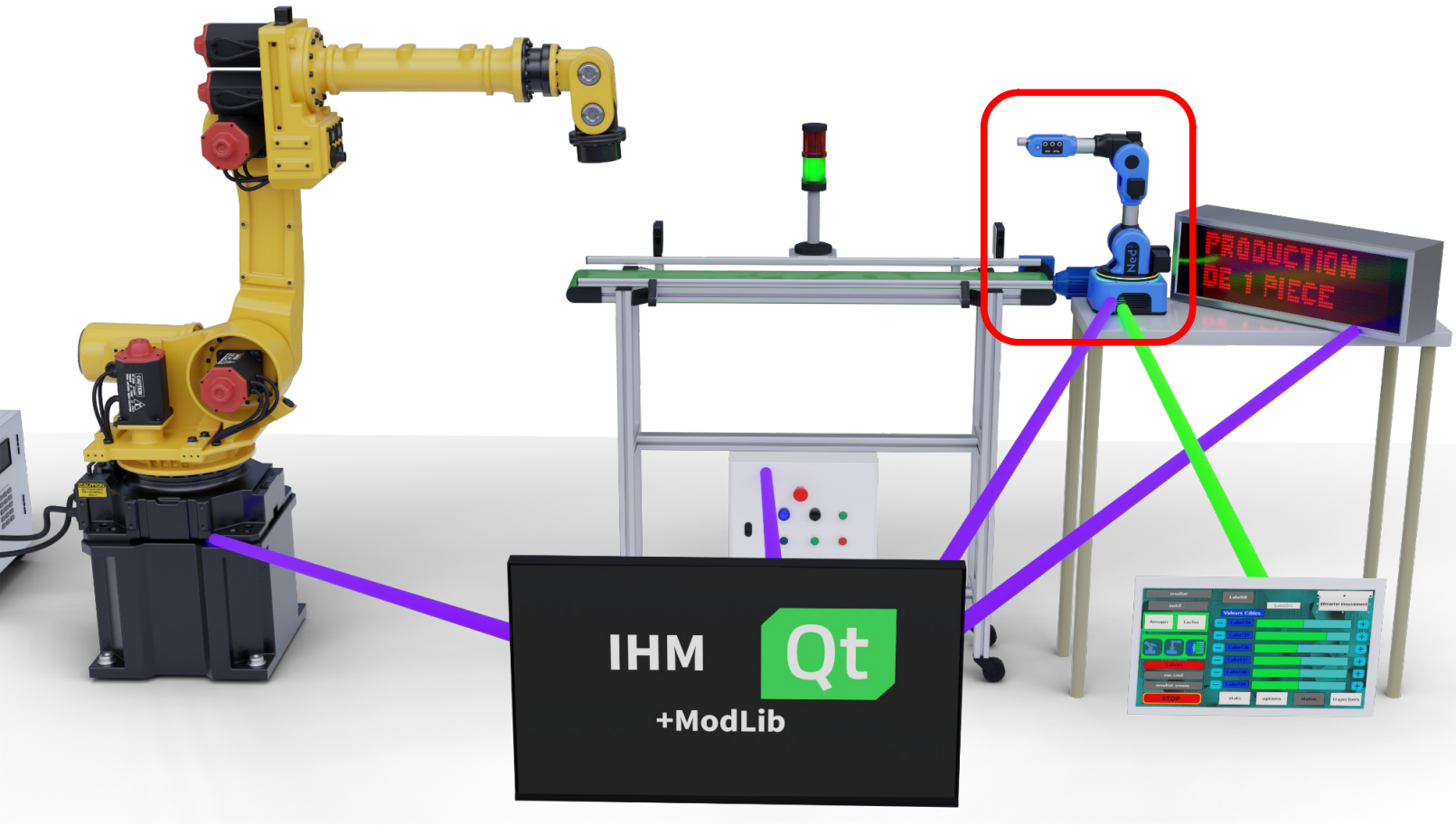


Figure 1: Vue 3D du projet

Ma partie est de programmer le robot **Niryo Ned 2** (plus d'informations dans le chapitre **Robot Niryo Ned 2**), qui se situe dans l'encadré rouge, afin qu'il puisse effectuer des mouvements, apprendre des enchaînements de trajectoire à suivre et de les restituer par l'intermédiaire de commandes via un registre **Modbus**, ainsi que d'être contrôlable via ce même registre **Modbus**.

2. Cahier des charges

Les tâches à remplir sont résumés ci-dessous, les miens étant les encadrés rouges.

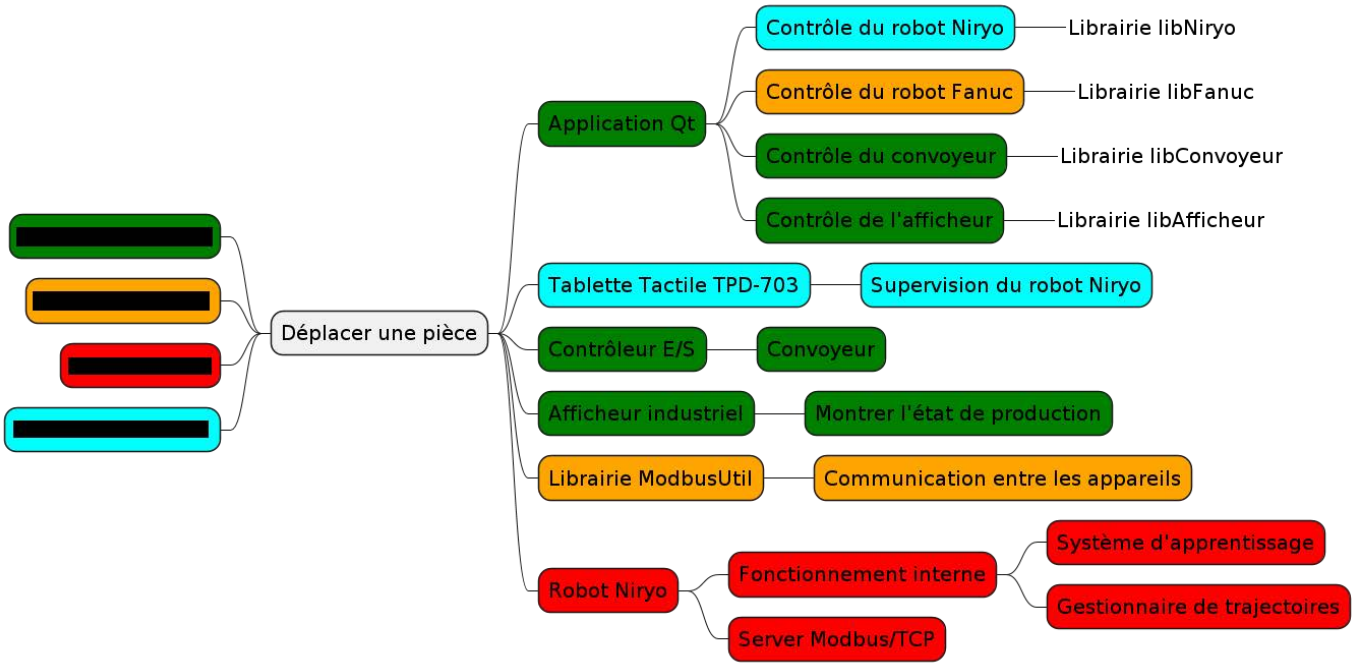


Figure 2: Diagramme répartition des tâches

Les tâches qui me sont attribuées, d'après le cahier des charges, sont donc les suivantes :

F5 : Contrôle – commande du robot 2 Niryo Ned 2

- Remise en conditions initiales, réinitialisation des axes
- Départ cycle, arrêt de la production
- Synchronisation du superviseur avec ce robot par des flags DI/DO (In/Out)
  - DI : In, flag d'attente de condition
  - D0 : Out, flag de signalement d'une condition
- La synchronisation peut se faire avec
  - le GPIO : Entrées, sorties directes sur le robot
  - le protocole Modbus/TCP
- Communication réseau avec le protocole Modbus/TCP en mode esclave
- Surveillance de l'état du robot, position des axes
- Gestion de plusieurs cycles différents de production

F8 : Base de données pour le stockage du plan mémoire Modbus/TCP du robot 2

- Base de données pour mémoriser les informations qui peuvent être lues ou écrites via Modbus/TCP
- Choix d'une base légère pour système embarqué
- Gestion d'un plan mémoire "Mots" et éventuellement d'un plan mémoire "Bits"
- Gestion des données pour stocker l'état du robot (auto, manuel), ses modes souhaités de marche et d'arrêt, les positions de ses axes, etc.

Pour des raisons, qui seront expliquées dans les **Solutions de conception**, la tâche **F8** n'a pas été réalisé.

Durant le projet, j'ai décidé de m'ajouter une partie dans la tâche **F5**, car lors de la conception logicielle, j'ai effectué des fonctionnalités supplémentaires qui n'étaient pas demandés initialement dans le cahier des charges.
Cette partie, ajoutée à la tâche **F5**, se situe ci-dessous :

- Fonctionnalités supplémentaires :
  - Système de logs par fichiers
  - Terminal de débogage pour les tests unitaires ou les techniciens
  - Architecture entièrement modulable
  - Format de fichier propre aux trajectoires
  - Gestion CLI pour modifier des composants, et pour les tests unitaires
  - Gestion avancée d'erreurs
  - Script de calibration avancée
  - Génération de diagrammes classes
  - Génération de documentation
  - Système anti double démarrage
  - Scripts de lancement et d'arrêt rapide

Pour faciliter la gestion du projet, nous avons utilisé un logiciel en ligne nommé **Trello**.
Grâce à cette solution, les diagrammes de Gantt peuvent être générés automatiquement à partir des tâches des membres, mais aussi la modification et l'assignation de tâches est plus simple.
Une image de notre gestion projet se situe en **Annexe 1**.

Sur la page suivante se trouve le diagramme de Gantt de mes tâches. (généré par **Trello**)

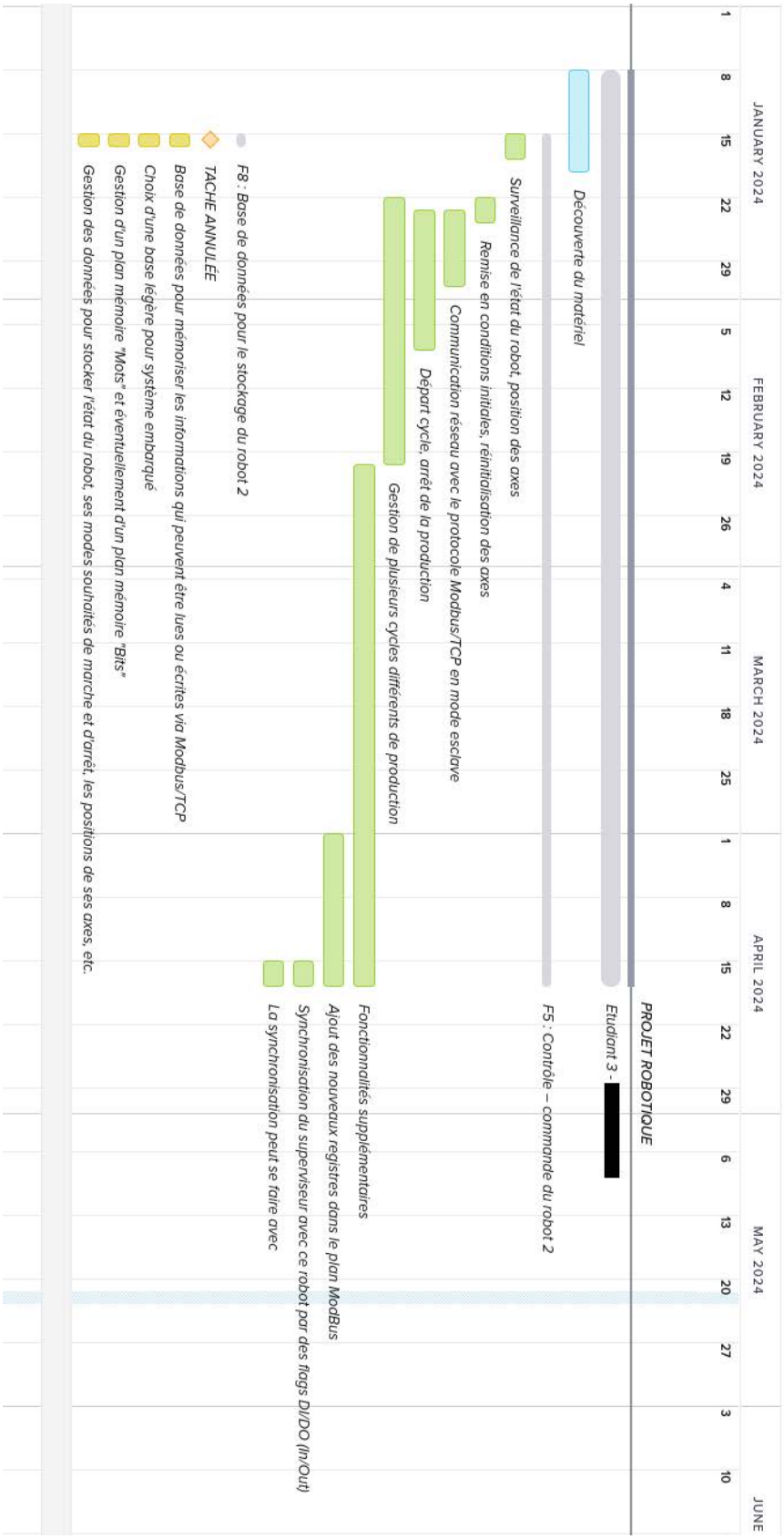


Figure 3: Diagramme de Gantt

3. Analyse

Le fonctionnement du système peut être représenté de cette manière :

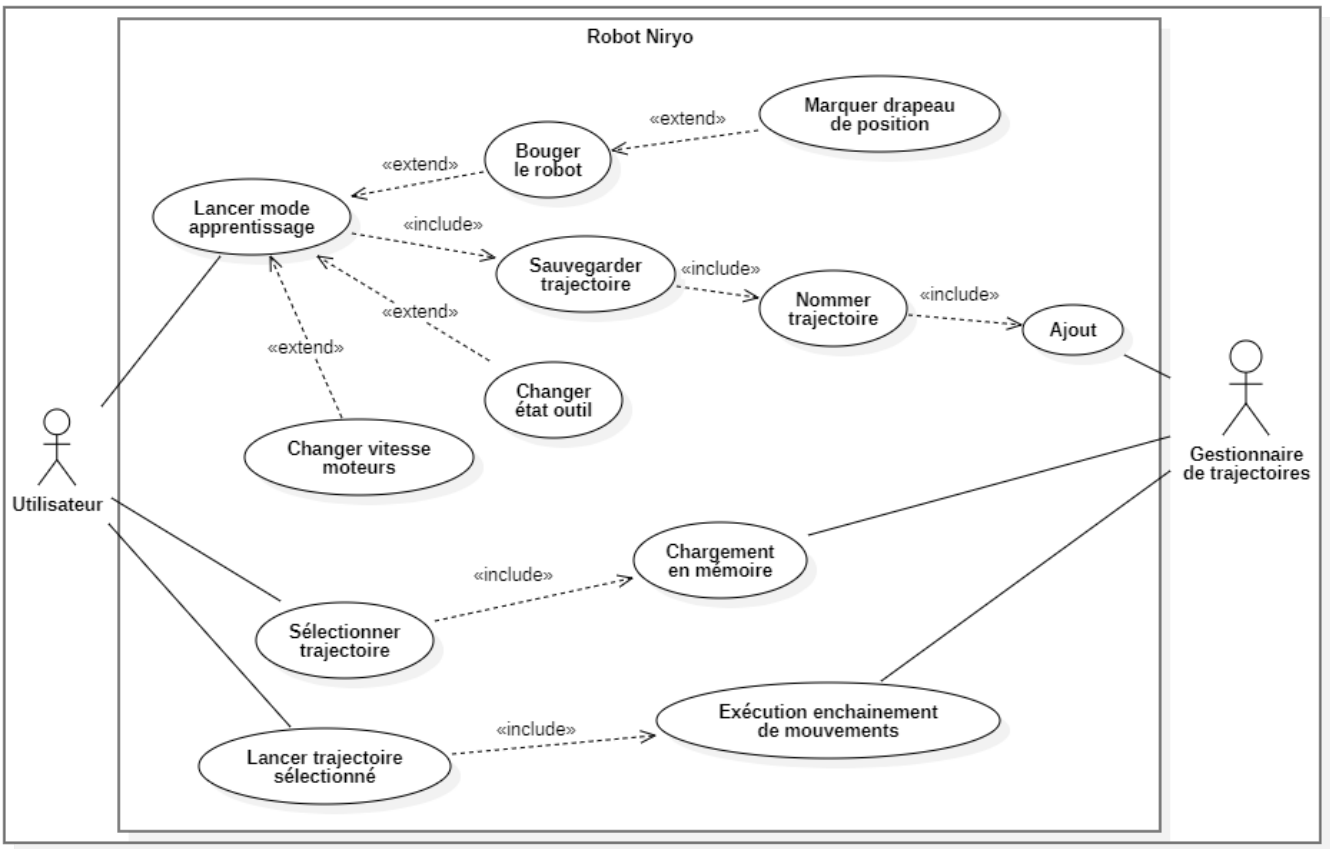


Figure 4: Cas d'utilisation Niryo

L'utilisateur, par l'intermédiaire des registres **Modbus**, peut lancer le mode d'apprentissage permettant d'enregistrer des séquences de mouvements, l'état de l'outil, ou bien la vitesse des moteurs entre chaque mouvement. Mais aussi sélectionner une trajectoire, contenant la séquence de mouvements, permettant de lire quelques informations à propos de cette dernière, comme son nom et son identifiant unique, mais aussi de la charger en mémoire et ainsi la lancer quand il le souhaite.
Et bien sûr, il peut lancer la trajectoire sélectionnée pour pouvoir effectuer l'enchaînement de mouvements.

Tout cela passe par l'intermédiaire du « gestionnaire de trajectoires », qui offre une interface entre les enchaînements de mouvements et un format de fichier, créé pour l'occasion (davantage d'informations dans le chapitre **Gestionnaire de trajectoires**), permettant de les sauvegarder sur un espace utilisateur.

4. Matériels et logiciels utilisés

4.1. Robot Niryo Ned 2

Le **Niryo Ned 2** est un bras robotique 6 axes assemblé en France et spécialement conçu pour accompagner des projets d'automatisation de postes et process. Il s'agit d'une solution complète et facilement reprogrammable pour aider les industriels dans l'automatisation d'un bout de leur chaîne de production.

Ce bras a également une vocation didactique et permet la découverte de la robotique, de la mécanique, de l'électronique, de la programmation et bien plus encore. Il offre aux enseignants la possibilité de réaliser des travaux pratiques présentant un intérêt et un plaisir d'apprentissage accrus aux étudiants. Le personnel éducatif et les étudiants peuvent ainsi reproduire des cas d'utilisations industrielles.

Le robot Niryo se compose de 7 parties et 6 articulations robotisées : La base, l'épaule, le bras, le coude, l'avant-bras, le poignet, ainsi que la main, offrent une liberté de mouvement à quasi 360 degrés.

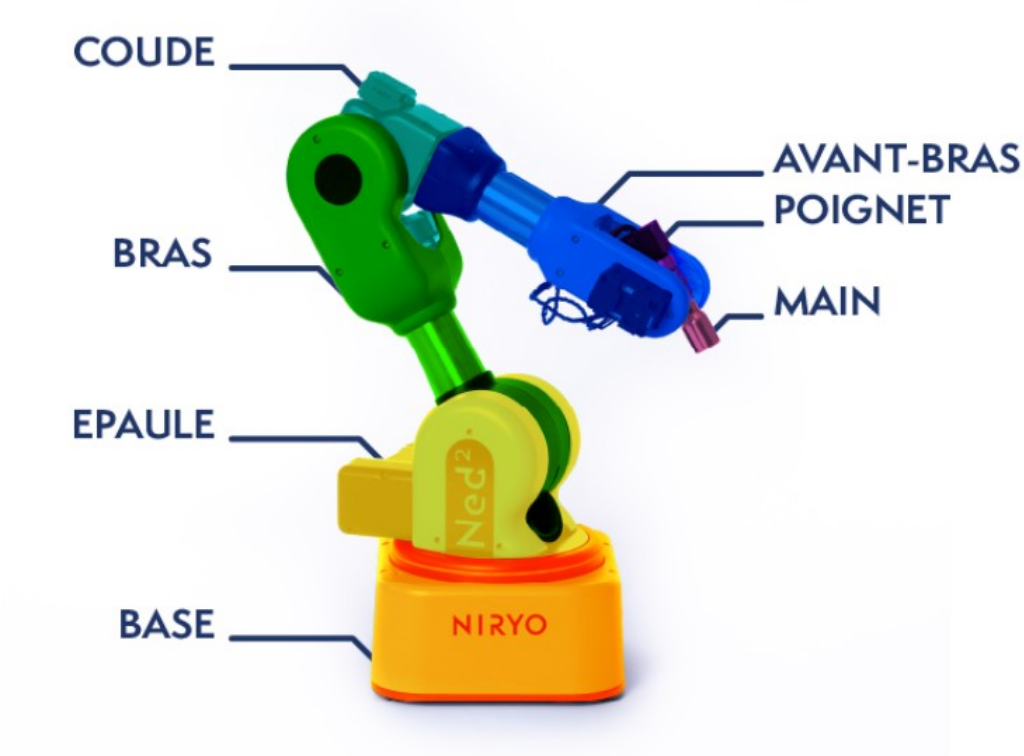


Figure 5: Vue d'ensemble Niryo

Il est aussi capable de placer le bout de la main à des coordonnées précises dans un espace 3D, grâce au système TCP (Tool Center Point). Ce dernier va automatiquement calculer et négocier la position des moteurs pour y mettre l'outil à l'endroit voulu.

4.1.1. Moyens de programmation

Le robot peut être contrôlé de différentes manières :

- **Niryo Studio** est le logiciel PC fournit avec, offrant une prise en main rapide du robot, donnant quelques informations sur ce dernier avec la possibilité de le mettre à jour automatiquement. Ainsi que des moyens ludiques de le contrôler, comme avec un langage de blocs ou un mode manuel avec des boutons.

- **RosWrapper** est la méthode principale à utiliser pour programmer le robot en **Python**. Elle utilise le système interne du robot et offre une librairie simple d'utilisation ainsi qu'un contrôle total. Cependant, les programmes doivent être stockés sur le robot pour pouvoir accéder à cette librairie.

- **PyNiryo** et **PyNiryo2** sont des librairies Python permettant l'envoie de commandes à distance. Elles sont équivalentes à la méthode de **RosWrapper**, mais à distance.

- **ROS** (Robot Operating System) est le système interne du robot, tous ces composants communiquent grâce à ce dernier. Il est possible de contrôler le robot à distance grâce à ce système, mais il en reste néanmoins assez dur d'emploi. Il est préférable de l'utiliser uniquement pour le système interne.

- **Modbus/TCP** est un vieux moyen de communication pour les machines. Il fonctionne par envoi de commandes à un appareil via la modification d'un registre. Cela sert, par exemple, effectuer des cycles de production. Dans ce cas, il permet contrôler la plupart des parties du robot, du moins, le principal.

- **MatLab** est un langage de script destiné au calcul numérique. Principalement utilisé pour l'analyse de données et des simulations, il est utilisé, dans ce cas, à la manière de ROS, et pouvant faire office de jumeau numérique (une simulation du robot qui bouge en temps réel avec le vrai).

4.1.2. Interface Matériel

L'architecture électrique du robot est composé de 5 parties, un diagramme explicatif est disponible en **Annexe 2**.
La première est le **Raspberry PI 4**, qui est le cerveau du robot, et offrant une interface de programmation en **Python**. La seconde est le **Niryo Shield**, permettant l'ajout des autres composants et fessant le pont entre tout le reste pour être contrôlable par le **Raspberry PI**. La troisième partie est le panneau arrière du robot, permettant de connecter les différents accessoires ainsi qu'un câble réseau, quelques prises USB et quelques entrées/sorties analogiques pour contrôler divers matériels. La quatrième se compose des différents moteurs du robot, représentant les axes, ainsi que quelques boutons et prises d'accessoires, sur le bout du bras, pour des interactions humaines. La dernière partie sert aux interactions humaines, un bandeau LED pour des effets visuels, ainsi que des haut-parleurs qui seront utilisés avec des sons, musiques ou une synthèse vocale.

Passons aussi un point rapide sur l'espace de travail du robot. Quelques diagrammes sont disponibles en **Annexe 3**. Il peut tourner a 350° sur l'axe horizontale, ainsi que 128° sur l'axe vertical, de plus, grâce à la main, il est capable d'accéder légèrement en dessous de sa basse.
Globalement, il est assez libre de mouvements et peut effectuer des tâches avec rescision grâce à ses moteurs.

4.1.3. Interface Logiciel

L'architecture logiciel se présente sous cette forme :

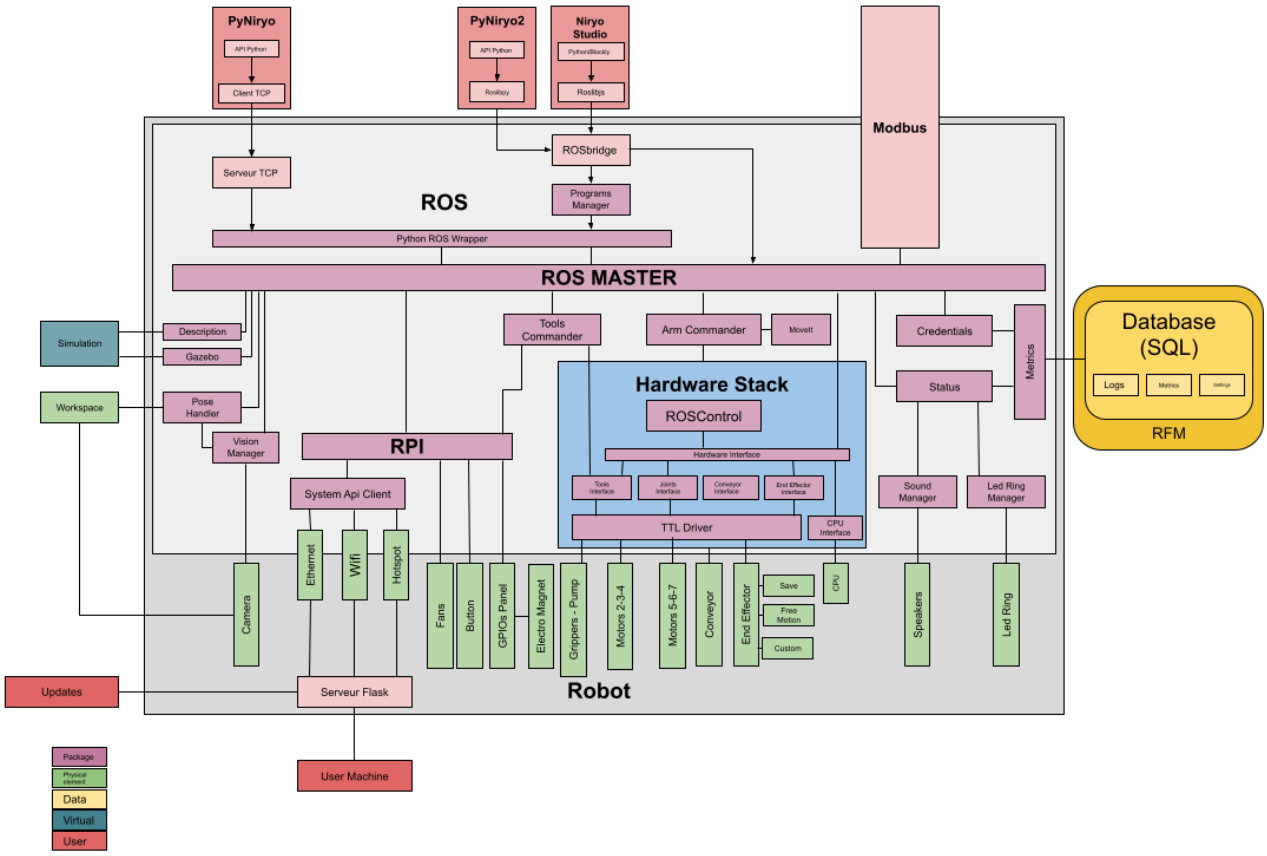


Figure 6: Architecture logiciel Niryo

Les zones en rouge, représente les interactions utilisateur possible, les zones en mauve, sont les parties logicielles du robot, celles en vertes, représente les parties matérielles. Et enfin, les zones en jaune et en bleu, sont les moyens de stockage des paramètres du robot et les parties virtuels.

En haut du diagramme, se situent les différentes méthodes de programmation du robot à distance, et le **« ROS Master »** est la zone interne de programmation du robot. Elle n'est normalement pas recommandée d'utilisation, mais c'est dans cette zone que j'ai programmé le robot pour effectuer les tâches demandées dans le cahier des charges.

En commentaire, je peux globalement dire que la partie logicielle du robot est assez bien conçu et très modulable.

4.2. Visual Studio Code

Figure 7: Logo VSCode

Visual Studio Code (VSCode) est un environnement de développement intégré (IDE) de **Microsoft**. Il est open-source et cross-platform et a été conçu pour les développeurs web, mais il prend en charge de nombreux autres langages de programmation tels que **C++**, **C#**, **Python**, **Java**, **etc**.

Personnellement, je l'utilise dans mon quotidien. Je le trouve très bien fait, et les extensions sont très fournies, donc je vais l'utiliser pour le projet.

4.3. Python3

Figure 8: Logo Python

Python est un langage de programmation puissant et facile à apprendre. Il dispose de structures de données de haut niveau et permet une approche simple, mais efficace de la programmation orientée objet (POO). Python est un langage idéal pour l'écriture de scripts et le développement rapide d'applications.

Étant donné que le robot se base sur une **Raspberry PI**, et que ce type de carte ce programme principalement en **Python**, j'ai donc choisi ce langage de programmation. C'est aussi, car j'ai une bonne connaissance de ce dernier, que je l'ai choisi.

5. Solutions de conception

5.1. Modification du cahier des charges

Comme évoqué dans **Cahier des charges**, la tâche concernant le stockage, de la position des axes du robot, ainsi que d'autres informations utiles, dans une base de données **SQL**, n'a pas été effectué. En effet, le robot est assez bien conçu et gère automatiquement la position de ses moteurs, ainsi qu'aux meilleures façons d'effectuer un mouvement.

De plus, il gère automatiquement la limite de ses axes, les collisions qui pourraient survenir pendant un mouvement, et bien d'autres sécurités. De ce fait, la programmation de tels systèmes n'est pas utile.

5.2. Méthode de programmation

Comme décrit dans le chapitre des **Moyens de programmation**, il y a plusieurs méthodes pour contrôler le robot. Mais celle que j'ai choisie est le **RosWrapper**, car certes, il faut que le programme soit dans le robot pour fonctionner, mais les temps de latence entre chaque commande est moindre. De plus, j'ai bien plus de contrôle sur ce dernier, puisque je peux interagir sur le système interne à tout moment.

Il me faut donc un accès à distance simple, pour pouvoir programmer sur le robot depuis une autre machine. J'ai alors utilisé une extension **VSCode** nommé **« Remote Development »**, me permettant de programmer sur une machine distante.

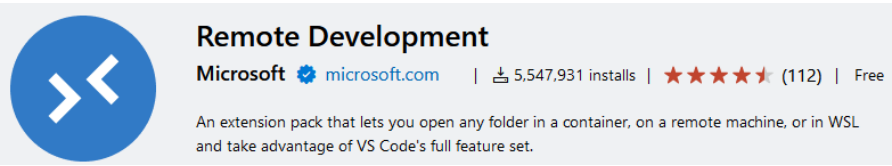


Figure 9: Logo extension vscode

5.3. Diagrammes de classes

Le diagramme de classe se trouve sur les pages suivantes.
Il résume grossièrement l'architecture du logiciel que j'ai créé, car je ne l'ai pas fait dès le début, il a été généré à la fin du projet avec un outil nommé **PyReverse**, puis remanié à la main.

> C'est un diagramme de classe extrêmement simplifié, et encore, je dois le couper sur deux pages, car il est trop grand.
>
> Un extrait du diagramme global peut être trouvé en Annexe 4.

5.4. Problème Modbus

Pendant la conception du logiciel, je n'avais pas remarqué que le serveur **Modbus/TCP** fournit par le robot n'allait pas couvrir tout le besoin.
Plusieurs solutions étaient possibles, pour pallier cela :

- Soit modifier le serveur existant en ajoutant les registres pour le besoin. Mais cela nécessite la modification de librairies interne au robot, et cela n'est pas très viable dans le futur, car une simple mise à jour pourra effacer tous les changements.

- Soit créer un deuxième serveur, qui contient les registres voulus (et plus si besoin). Mais cela nécessite deux connexions différentes à utiliser pour un client, donc, l'utilisation d'un deuxième port.

C'est cette dernière solution qui a été choisi, car plus simple à mettre place et est plus viable pour le futur. Le seul défaut étant que les clients souhaitant contrôler le robot, doivent utiliser le serveur ajouté pour apprendre et lire des trajectoires au robot (ce qui correspond au besoin), et utiliser le serveur du robot pour faire tout le reste.

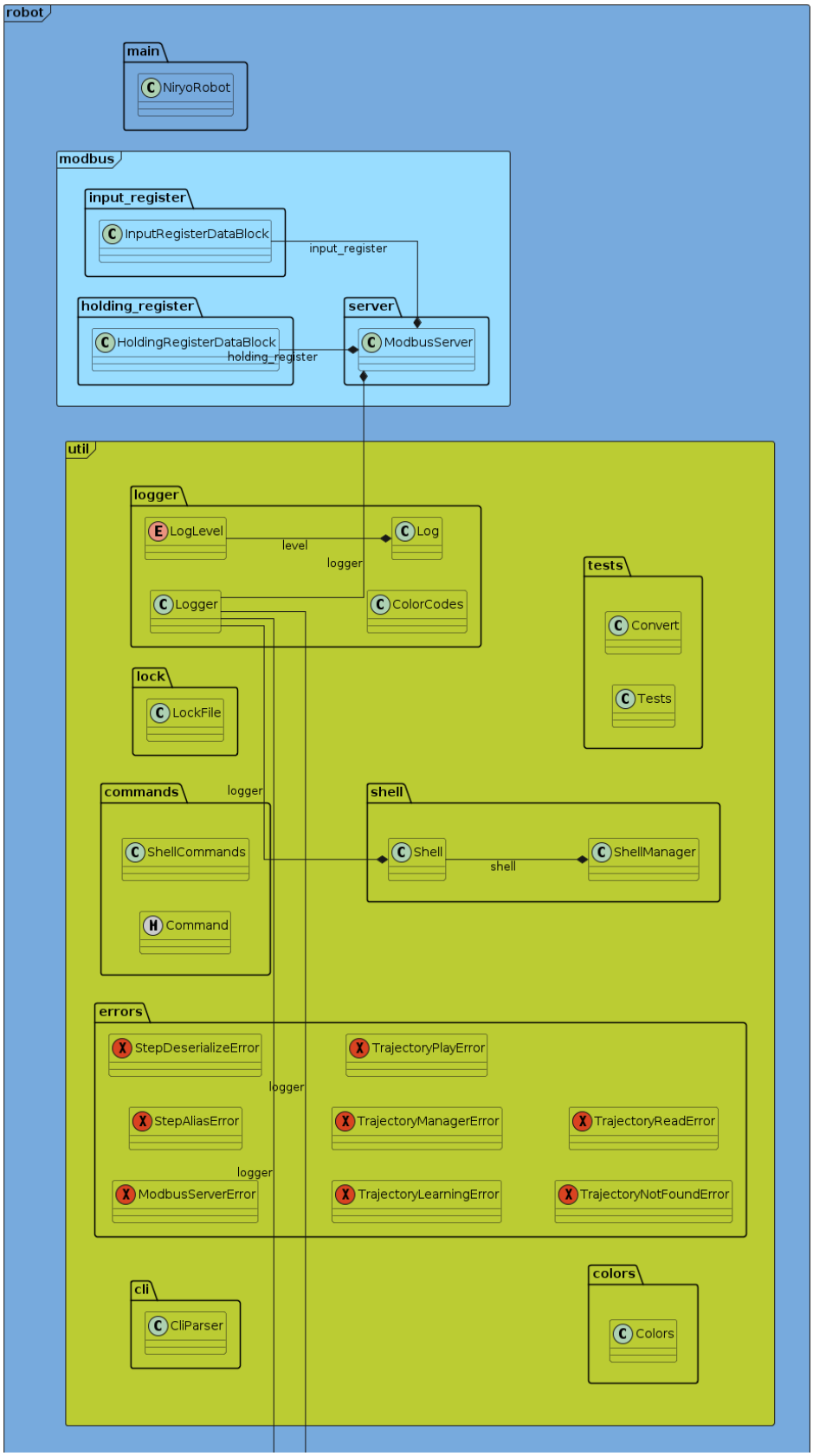


Figure 10: Diagramme de classes simplifié – partie 1

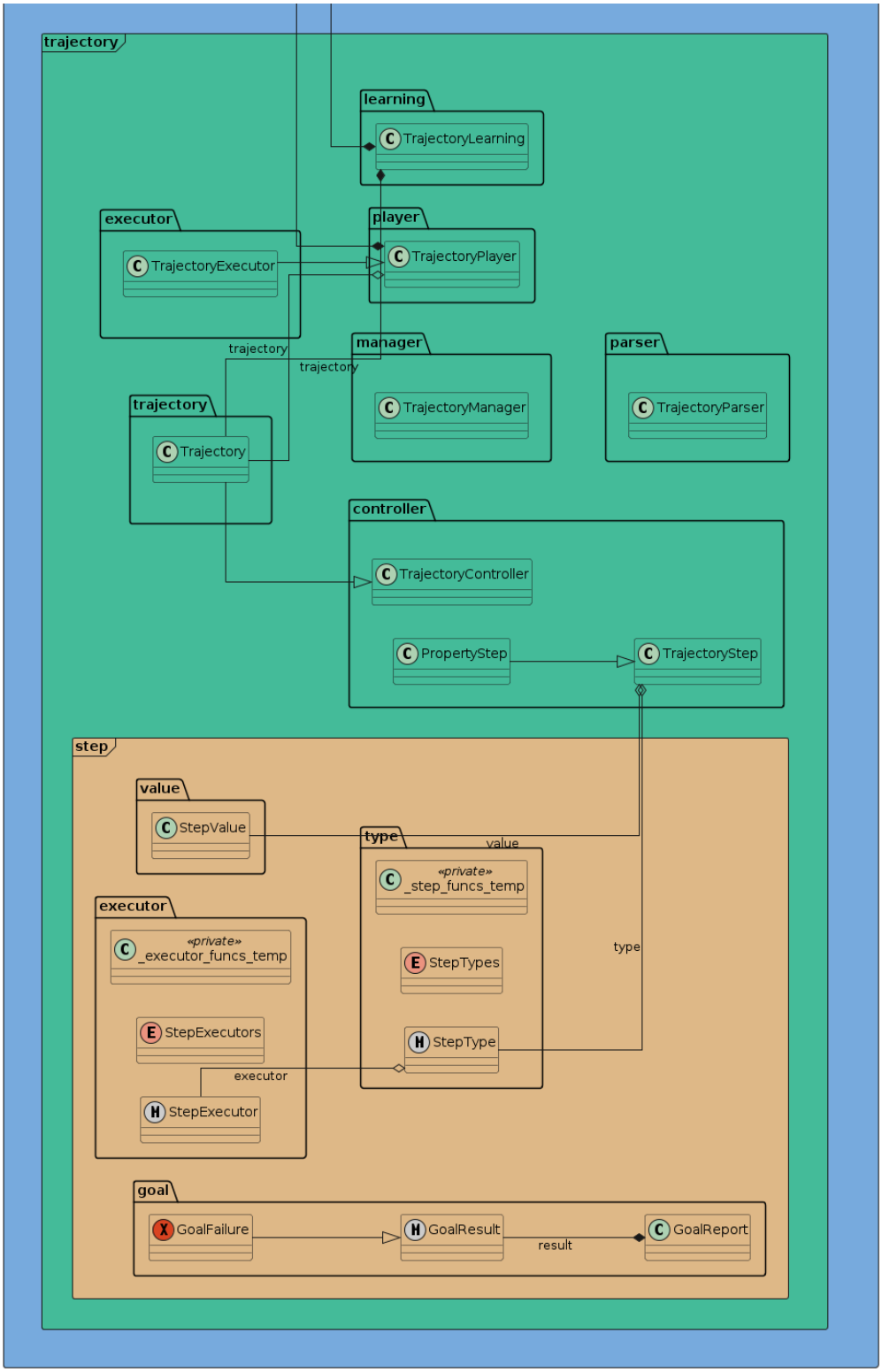


Figure 11: Diagramme de classes simplifié – partie 2

6. Réalisation

6.1. Prise en main du robot

Comme dit précédemment, j'ai donc programmé le **Niryo** à distance avec **VSCode**. Mais pour commencer, je l'ai pris en main grâce au **Niryo Studio.** Quelques scripts **Python** ont été effectués pour le faire bouger, et chanter. Et cela est plutôt satisfaisant et intuitif à utiliser.
Le seul défaut étant qu'il faut un compte pour utiliser le logiciel.

> Cela peut être contourné en ajoutant la ligne **« 127.0.0.1 api.niryo.com »** au fichier **/etc/hosts** .

Ci-dessous une capture d'écran du logiciel :

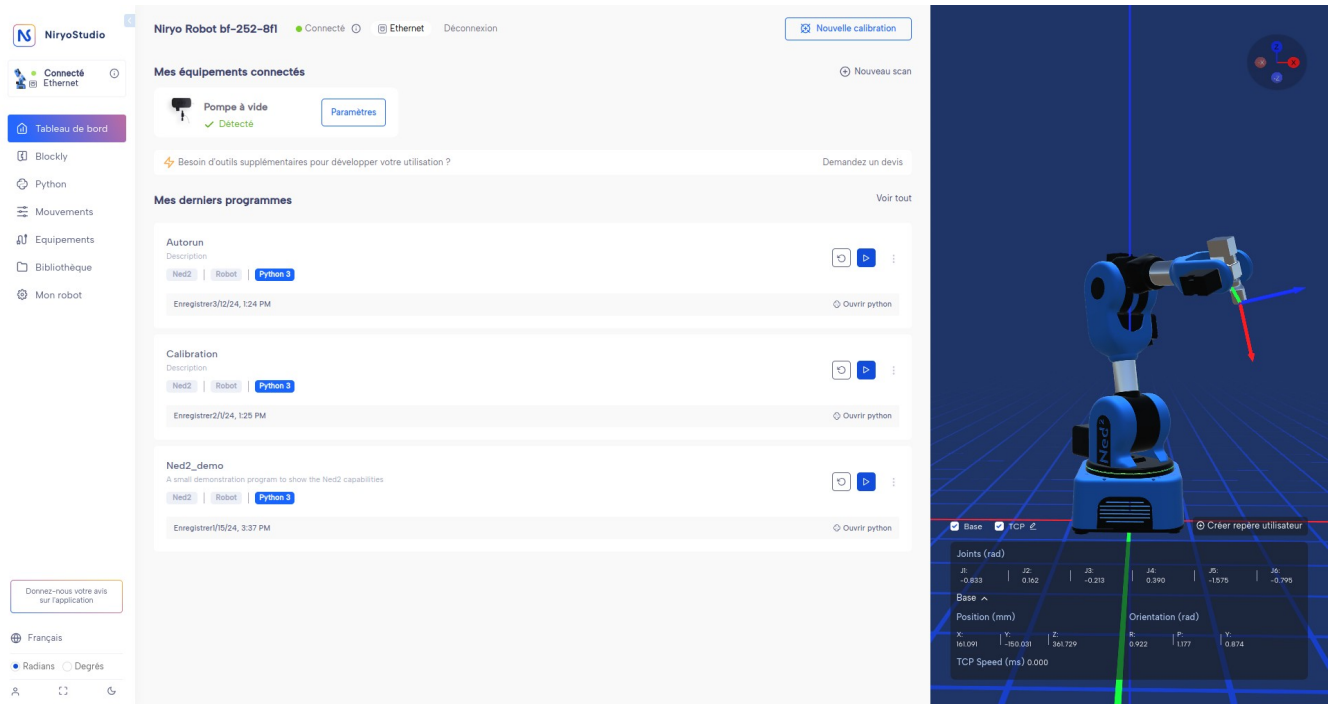


Figure 12: Capture d'écran Niryo Studio

Le studio est très complet, avec même une visualisation 3D en temps réel du robot, ainsi que la position de chaque axe, à droite. Le tout est découpé en plusieurs onglets.

Des onglets pour programmer en **Blockly**, un langage avec lequel il faut déplacer des blocs pour faire un programme, un onglet **Python** pour le contrôler depuis le studio. Il y a aussi des onglets pour bouger manuellement le robot et gérer l'outil qui y est connecté. Ainsi qu'un onglet pour voir les trajectoires enregistrées dans le robot, les musiques, les sons, les zones de travail et les points repères. Et bien sûr, un onglet pour avoir quelques statistiques sur le robot, comme la température de chaque moteur ou les logs.

Toutes les fonctionnalités ont été testées et elles sont assez diverses. Par exemple, le robot est capable de jouer de la musique ainsi que de parler grâce à une synthèse vocale, mais cette fonctionnalité requiert l'accès à Internet.

6.2. Premier programme

Ensuite, je suis rapidement partie en accès à distance, pour mieux apprendre à utiliser le robot avec le programme de démo qui était fourni avec.

Viens l'étape de la gestion des trajectoires. Le robot possède une gestion de trajectoire, avec des enchaînements de mouvement, cependant après la prise en main avec le **Niryo Studio**, ce système ne prend pas en charge les changements d'état de l'outil.
Dans tous les tutoriels sur internet, la procédure est de créer une trajectoire, changer l'état de l'outil, puis d'en créer une autre. Cela peut être une solution, mais cette dernière nécessite donc la création d'un script **Python**, pour chaque trajectoire.

L'objectif est alors, maintenant, de créer un nouveau système de trajectoire plus modulaire. Avec la possibilité d'ajout de type d'étapes supplémentaire, comme la modification de la vitesse entre chaque mouvement ou encore changer la couleur du bandeau LED.

J'ai donc fait un premier programme qui me permet de contrôler de robot à partir du clavier, d'enregistrer les étapes, et de les reproduire par l'appui d'un bouton sur le robot.
Ce dernier fonctionne parfaitement et me donne envie de continuer sur cette solution, avec bien sûr, de l'optimisation.

> Le code du programme se trouve en **Annexe 5**.

6.3. Architecture logicielle

Après plusieurs mois de programmation, l'architecture logicielle est quasiment terminé. Tous les fichiers sont en place, manquant plus que le serveur **Modbus** à être codé.
Elle a été conçue pour être modulable, lisible, et débuggable, car tout ce qui doit être écrit dans le terminal, doit passer par un logeur qui enregistre tout dans un dossier de logs. En plus de donner de jolies couleurs au texte.

Un diagramme ci-joint, montre l'arborescence des fichiers du projet.

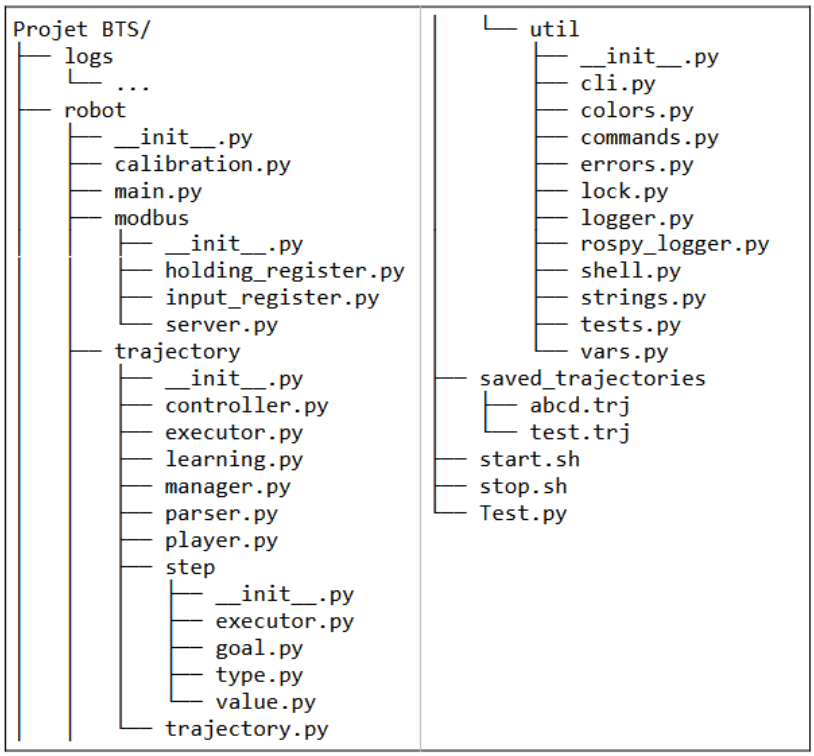


Figure 13: Arborescence des fichiers

Ici nous avons une construction par rôle logique. Le dossier **« util »** ne contient que des fichiers utilitaires, soit des librairies qui tiennent en un seul fichier ou ne dépendent que de fichiers de ce même dossier. Le dossier **« modbus »** contient tout ce qui est en rapport avec **Modbus**, etc.
Les classes sont aussi rangées, dans des fichiers, nommés par type de fonction.

6.4. Gestionnaire de trajectoires

Le gestionnaire de trajectoires est une partie essentielle et est la plus longue à programmer.
Ce système représente plus d'une dizaine de fichiers dans le dossier **« trajectory »**, vue dans la **Figure 13**.

Elle se compose d'étapes (dossier **« step »**), d'un contrôleur, d'un exécuteur, d'un analyseur (pour la gestion du format de fichier), d'un joueur (pour l'exécution de l'enchaînement de mouvements), d'un apprenant (pour l'enregistrement des mouvements) et bien sûr de la trajectoire contenant tous les types d'actions. Ce dernier permet de créer et récupérer des actions de manière simple. Son code se situe en **Annexe 6**.

6.4.1. Les actions

Un petit zoom sur les différents types d'actions que l'on peut enregistrer dans une trajectoire, en plus de faire des enchaînements de mouvements.

Une action se compose d'une commande, uniquement d'un seul caractère, ainsi que d'arguments.

```
class StepType:
  def __init__(self, id: int, alias: str, label: str, needed: bool, is_property: bool,
               duplicate_allowed: bool, none_value_allowed: bool,
               executor: StepExecutor, serializer: 'function', deserializer: 'function'
  ):
```

Figure 14: Définition d'une étape de trajectoire

Un type d'action ce construit sous cette forme :

- Un identifiant
- Un alias, le nom de la commande, composé d'un seul caractère uniquement
- Un nom, pour reconnaître l'étape plus facilement
- Requise, définie si cette action doit être présente dans la trajectoire
- Propriété, cette action ne servira pas dans la trajectoire, uniquement pour définir des choses à propos de cette dernière, comme un nom
- Duplication autorisée, si activé, l'action ne peut pas être présente plusieurs fois
- Valeur nulle autorisée, permet d'avoir une commande sans argument
- Un exécuteur, cela définie ce qu'il doit être effectué quand l'action est exécutée
- Un sérialiseur, qui définie sous quel format les arguments doivent être stockées.
- Un dé-sérialiseur, qui permet de retrouver les arguments formatés par le sérialiseur.

Voici les différentes actions programmés :

```
class StepTypes(enum.Enum):
    label = StepType(0, 'l', "label", True, True, False, False,
        StepExecutors.none.value, _step_funcs_temp.serialize_str, _step_funcs_temp.deserialize_str)
    id =  StepType(1, 'i', "id", True, True, False, False,
        StepExecutors.none.value, _step_funcs_temp.serialize_int, _step_funcs_temp.deserialize_int)
    start = StepType(2, 's', "start point", True, True, False, False,
        StepExecutors.start.value, _step_funcs_temp.start_serializer, _step_funcs_temp.start_deserializer)
    move = StepType(3, 'm', "movement", False, False, True, False,
        StepExecutors.move.value, _step_funcs_temp.move_serializer, _step_funcs_temp.move_deserializer)
    tool = StepType(4, 't', "tool", False, False, True, False,
        StepExecutors.tool.value, _step_funcs_temp.tool_serializer, _step_funcs_temp.tool_deserializer)
    acceleration = StepType(5, 'a', "motor acceleration", False, False, True, False,
        StepExecutors.acceleration.value, _step_funcs_temp.serialize_int, _step_funcs_temp.deserialize_int)
    file = StepType(6, 'f', "file name", False, True, False, False,
        StepExecutors.none.value, _step_funcs_temp.serialize_str, _step_funcs_temp.deserialize_str)
    none = StepType(255, 'u', "<unknown>", False, False, True, True,
        StepExecutors.none.value, lambda *_: '', lambda *_: None)
```

Figure 15: Liste des actions de trajectoire

Nous avons des actions dites de propriétés, qui ne servent qu'à définir des informations dans la trajectoire (leur ordre dans le fichier n'a pas d'importance) : le nom de la trajectoire (**« label »**), un identifiant unique (**« id »**), la position de démarrage (**« start »**) et le nom du fichier (**« file »**).

Et nous avons et les actions dites utiles, qui définissent les choses à faire, dans cette trajectoire, dans l'ordre séquentiel (leur ordre a donc une importance, dans le fichier) : le mouvement qui contient la position des moteurs à atteindre (**« move »**), l'état de l'outil (**« tool »**), ainsi que la vitesse des moteurs entre chaque mouvement (**« acceleration »**).

Cette architecture est en conséquence très modulaire, il suffit de créer une action, de lui donner les propriétés voulues, ainsi que les moyens d'exécution et de codage / décodage. Et le reste est automatique.

6.4.2. Format de fichier

Tout cela doit être sauvegardé dans un fichier et pouvoir être restituer en format objet pour l'utilisation dans **Python**. C'est le rôle du **« parser »** qui va écrire toutes les actions de la trajectoire dans un fichier.

```python
@staticmethod
def write(traj: Trajectory):
    file = traj.get_filename()
    if file: file = get_trajectory_file(file)
    else:
        # Format a new file name with trajectory name (or default name), if not defined
        name = traj.get_name()
        if not name: name = "untitled-"+str(traj.get_id() or "trajectory")
        file = TrajectoryParser.make_trajectory_file_by_name(name)
        traj.set_filename(get_child_path(file))

    content = traj.get_content()
    path = get_parent_path(file)
    if not os.path.exists(path): os.mkdir(path)
    with open(file, "wt") as f:
        for t in content: f.write(t.serialize() + '\n')
```

Figure 16: Fonction d'écriture du parser

Cette fonction de la classe **TrajectoryParser**, va lister les actions (la vérification de leurs validités est faite avant), les formatés à l'aide du sérialiseur et les écrire dans un fichier au format **.trj**, dans un dossier spécifique.
Elle va aussi mettre un nom par défaut à la trajectoire, si elle n'en possède pas.

```python
@staticmethod
def read(file: str) -> Trajectory:
    traj = Trajectory()
    traj.set_filename(get_child_path(file))
    found_steps = []

    with open(file, "rt") as f:
        i = 1
        line = TrajectoryParser.read_next_step(f)

        while line:
            step = TrajectoryParser.find_step_type(line)

            if not step.duplicate_allowed and step in found_steps:
                raise TrajectoryReadError(file, f"duplicated {step.label} step ({step.alias!r}) at line {i}. "
                                                "This step must only appear once in this trajectory.")
            try:
                if step.is_property: traj.set_property(PropertyStep(step, step.deserialize(line)))
                else: traj.add_step(TrajectoryStep(step, step.deserialize(line)))
            except Exception as e:
                raise TrajectoryReadError(file, f"unable to decode {step.label} step ({step.alias!r}) at line {i}: ", e) from None

            found_steps.append(step)
            i += 1
            line = TrajectoryParser.read_next_step(f)

    if not all(s in found_steps for s in TRAJECTORY_NEEDED_STEPS):
        steps = ", ".join(f"{s.alias!r} ({s.label})" for s in TRAJECTORY_NEEDED_STEPS if s not in found_steps)
        raise TrajectoryReadError(file, "missing needed step(s) in trajectory: " + steps)
    return traj
```

Figure 17: Fonction d'écriture du parser

Cette fonction fait donc l'inverse, elle lit les étapes une par une et tente de les décoder à l'aide du désérialiseur. Elle reconstruit alors l'objet de trajectoire avec ce qui est décodé et renvoie des erreurs ce qui n'a pas pu être décodé.

6.5. Serveur Modbus

Comme mentionné dans le chapitre sur le **Problème Modbus**, l'utilisation d'un deuxième serveur **Modbus/TCP** est donc requis. Le serveur déjà existant étant sur le port 5020, le deuxième a été placé sur le port 5022.
La construction du nouveau serveur a été simple, car il suffisait de copier-coller celui existant et modifier quelques informations sur le serveur ainsi que le port, et programmer les classes qui servent de « input regiter » et « holding register ». La structure de l'existant a globalement été gardée.

Voici un résumé des registres **Modbus** disponibles, qui ont beaucoup été utilisés par ******.

Holding register:
0 : Mode apprentissage (1 : on, 0 : off) (Si relancé sans sauvegarde, réinitialise les infos de trajectoire en cours, sinon sélectionne automatiquement la trajectoire)
1 : Marquer un drapeau (1 : valider le drapeau)
2 : Changer le status de l'outil(1 : activé, 0 : désactivé)
3 : Vitesse des moteurs (1-200)

10-30 : Nom trajectoire sélectionné (Le nom doit être obligatoirement être fournis en format ASCII et composé uniquement de caractère visible)
31 : ID trajectoire sélectionnée

40 : Sauvegarder / Supprimer trajectoire (2 : supprimer, 1 : sauvegarde, 0 : rien) (Une fois sauvegardé ou supprimé, la trajectoire est désélectionnée)
41 : Sélectionner index trajectoire
42 : lancement trajectoire (DI/DO) (1 : lancer / en cours, 0 : terminé)
43 : Stopper la trajectoire en cours (1 : stopper)

50 : Nombre de trajectoires enregistrées
51 : Une trajectoire est sélectionnée (1 : oui, 0 : non / désélectionner)

100 : Status robot et Codes d'erreur (0-9 : même que le robot, … <voir codes d'erreur>)

Codes d'erreur :
99 : Erreur inconnue, la consultation des logs sera nécessaire pour connaitre l'erreur
100 : Mode apprentissage est activé
101 : Mode apprentissage non activé
102 : Pas de trajectoire sélectionnée
103 : La trajectoire n'existe pas
104 : Nom de trajectoire contient des caractères invalides
105 : Nom de trajectoire vide
106 : Trajectoire invalide, possède des éléments manquant ou invalide
107 : La trajectoire n'est pas en cours d'exécution
108 : La trajectoire est déjà en cours d'exécution
109 : La trajectoire a été annulée

Figure 18: Liste des registres Modbus

6.6. Terminal technicien

La création d'un terminal de commande, a été effectué. C'est un système très minimaliste qui n'inclut pas tout ce qu'un terminal classique peut faire, mais cela est suffisant.
Le terminal se lance dans nouveau programme pour fonctionner correctement.

Voici la liste des commandes qui ont été implémentés pour pouvoir contrôler le robot.
C'est une liste plutôt courte, mais elle permet de faire la majorité des choses qu'un technicien pourrait avoir besoin.

```
commands: 'list[Command]'  = [
  Command("help", "Print this help", help),
  Command("exit", "Exit the main program", lambda *_: 255),
  Command("loglevel", "Control verbosity level of logging", loglevel, "[level]"),
  Command("manual", "Control the robot manually by keyboard", manual),
  Command("deep", "Set/Get deep learning mode", deep, "[on|off]"),
  Command("default", "Move robot to default position", default),
  Command("halt", "Stop the current robot movement", halt, "[deep]"),
  Command("clear", "Clear collision detected trigger", clear),
  Command("calibrate", "Start new robot calibration", calibrate),
  Command("play", "Play a trajectory or list it", play, "[reload|type-<name|id>]"),
  Command("exec", "Execute python code in runtime", exec_, "<code...>", 1),
  Command("speed", "Set/Get arm max speed (velocity) between 1 and 200%", speed, "[percent]")
]
```

Figure 19: Liste des commandes du terminal

6.7. Système de logs

Un système de logs a aussi été fait, il est assez simple d'utilisation avec la création d'un objet Logger, ainsi qu'un sujet. Il permet d'avoir un affichage agréable dans le terminal, avec l'utilisation de couleurs, ainsi que des types de logs (comme les informations, les erreurs, etc).

Ce système introduit aussi nativement, l'écriture des logs dans les fichiers, ce qui permet de garder une trace de ce qui a été écrit, ainsi que d'analyser de potentiels problèmes.
Voici une image des logs dans le terminal, au démarrage du robot :

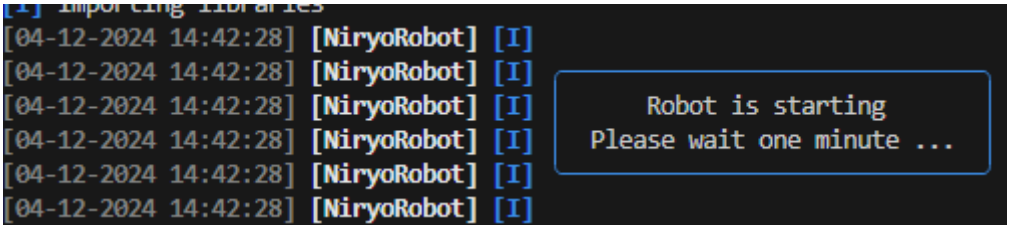


Figure 20: Logs terminal démarrage Niryo

6.8. Système anti double démarrage

Pour éviter le lancement de plusieurs instances du programme, le système d'anti double démarrage, va notifier l'autre programme de son lancement, grâce à un système de lockfile.
Un programme va écrire un fichier spécifique et l'écouter en permanence, si ce dernier est demandé à l'écoute, c'est qu'un autre programme cherche à l'utiliser et donc probablement le même.
Grâce à ce système, peut "communiquer" avec un autre pour, par exemple, lui demander de s'arrêter.

Dans notre cas, si le même programme est relancé, l'ancien va notifier l'utilisateur avec un signal sonore, puis s'arrêter correctement avant de donner la main au nouveau.

6.9. Arguments de ligne de commande

Il y a aussi un système de ligne de commande permettant de désactiver certaines parties du programme, comme par exemple, ne pas faire de calibration automatique ou ne pas lancer le terminal. Cela est assez pratique pour les tests unitaires ou lancer le programme d'une manière différente en fonction des situations.

Ci-contre, la liste des arguments possible, quand on lance le script de démarrage.

```
usage: main.py [OPTIONS]

options:
  -h, --help            show this help message and exit
  --no-calibration      disable auto calibration at start of program
  --no-trajectories     do not load trajectories
  --no-modbus           disable Modbus/TCP server
  --no-terminal         disable the terminal
  --no-status-sound     disable colors when logging
  --no-default-pose     do not move to default pose when stopping robot
  --no-colors           disable colors when logging
  --no-write-logs       disable writting of logs in files
  --logs-path LOGSPATH  the directory to write logs. (default is working directory)
  --log-folder NAME     the folder name to store log files. (default is 'logs')
  --log-level {debug,info,warn,err,none}
                        set a default logging level

Note: Logs writing will be done into files, split every 512 KB.
```

Figure 21: Liste des arguments de lancement

6.10. Génération d’un wiki

En petit supplément, un wiki de toutes les fonctions, classes et fichiers, de ce que j’ai fait, a été généré. J’ai utilisé un outil nommé **PyDoctor**, qui va parcourir les fichiers à la recherche de code **Python** et extraire les noms de fonctions, ainsi que leurs paramètres et la documentation associé.

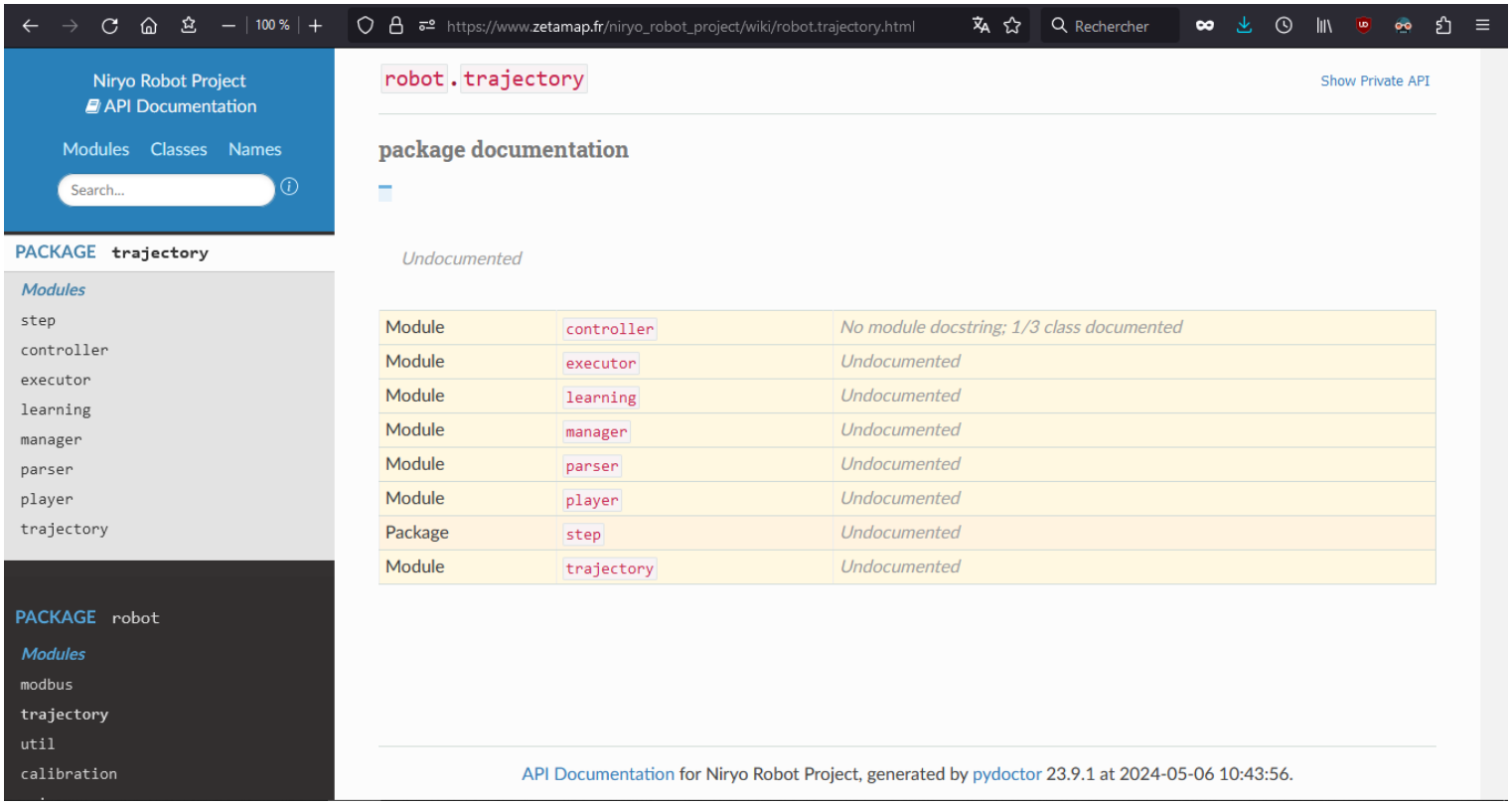


Figure 22: Capture d'écran du wiki

Tests unitaires

Code des tests unitaires en **Annexe 7**.

Test fonctionnel :	**Programme de démo du Niryo Ned 2**		
Objectif :	Vérification du système pour la mise en vente		
Initialisation :	Lancer le programme **Test.py** du projet et connecter le robot à internet		
Scénario :			
ID	**Démarche**	**Comportement attendu**	**Validation**
1	Vérification de la synthèse vocale	Le robot parle	OK
2	Vérification de la connexion internet	Le robot peut envoyer le rapport de tests	OK
3	Vérification du matériel (moteurs, capteurs, entrées/sorties, I2C)	Pas de message d'erreurs	OK
4	Test de calibration	La calibration s'effectue correctement	OK
5	Test des haut parleurs (volume sonore, sons)	Les sons sont joués	OK
6	Test du bandeau LED (tous les modes d'affichage)	Pas de message d'erreur	OK
7	Vérification des limites des moteurs	Pas d'erreur moteur, de message d'erreur	OK
8	Test des spirales	Pas de blocage ni d'erreurs	OK
9	Test de divers mouvements complexes	Pas de blocage, tous les mouvements sont effectués	OK
10	Test du « pick and place »	Pas de blocage ni d'erreurs	OK
Commentaire : **Tout s'exécute correctement**		Approbation :	

Test fonctionnel :	**Programme principal du projet**		
Objectif :	Vérification de fonctionnement pour le projet		
Initialisation :	Lancer le programme principal avec les arguments associés		
Scénario :			
ID	**Démarche**	**Comportement attendu**	**Validation**
1	Initialisation des librairies	Message de log notifiant l'importation	OK
2	Connexion au robot	Pas de message d'erreur	OK
3	Vérification du lockfile	Pas d'avertissements	OK
4	Calibration avancée	Pas de message d'erreur	OK
5	Chargement des trajectoires	Pas de message d'erreur	OK
6	Lancement du serveur Modbus/TCP	Pas de message d'erreur	OK
7	Lancement du terminal technicien	Pas de message d'erreur	OK
Commentaire : **Tout s'exécute correctement**		Approbation :	

Conclusion

Pour conclure, j’ai trouvé ce projet très passionnant. J’ai pu perfectionner mes connaissances en programmation, me faire des bases pour créer des architectures logicielles et construire mes programmes plus structurés et modulaires.
Cela m’a aussi aidé sur la communication et la compréhension des API que je fournis aux autres pour être utilisés. Comme avec ******, qui devait utiliser mon serveur **Modbus/TCP**.

Globalement, j’ai bien aimé et cela m’encourage à continuer sur la voie de la programmation logicielle.

Sitographie

Description du robot : https://www.fsi-france.fr/produit/nyrio-ned-2/

Manuel d'utilisation : https://archive-docs.niryo.com/product/ned2/v1.0.0/generated_pdfs/pdf_fr.pdf

Schéma électrique : https://archive-docs.niryo.com/product/ned2/v1.0.0/fr/source/hardware/electrical_interface.html#electrical-interface-overview

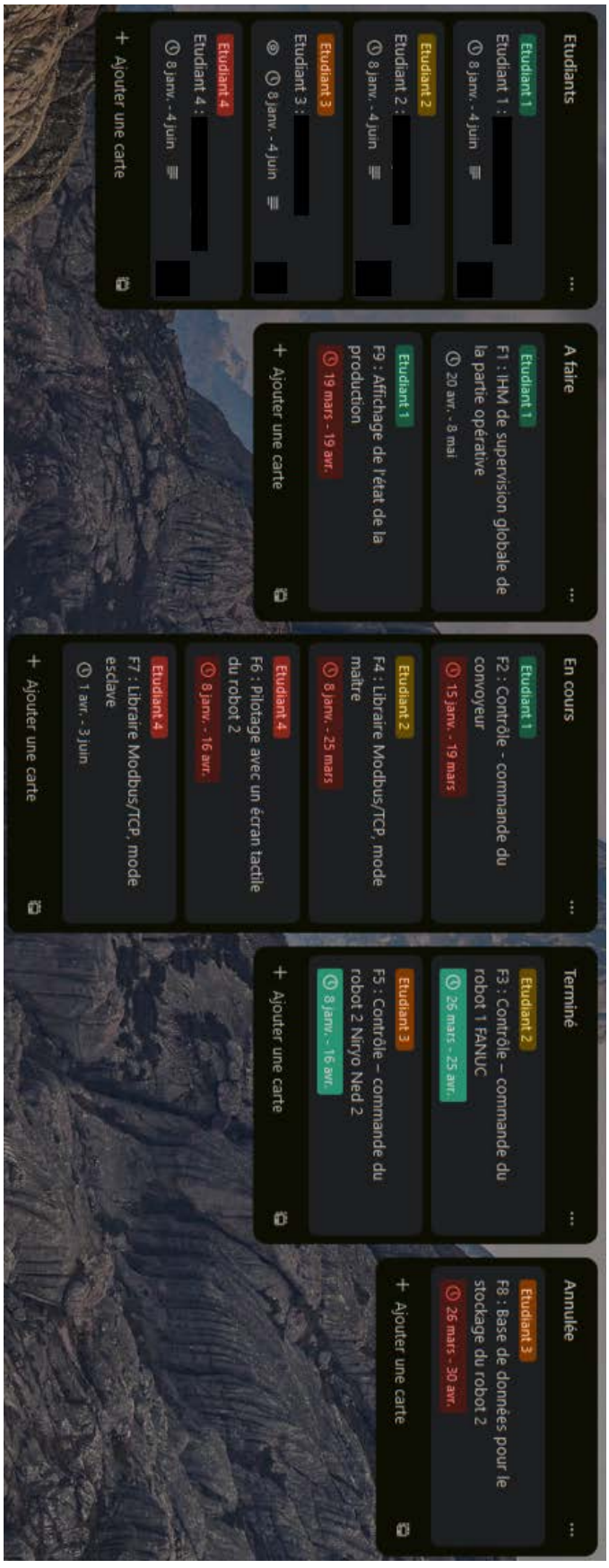
Schémas de l'espace de travail : https://archive-docs.niryo.com/product/ned2/v1.0.0/fr/source/hardware/mechanical_interface.html#robot-workspace

Schéma architecture logiciel : https://github.com/NiryoRobotics/ned_ros?tab=readme-ov-file#ned-ros-stack-overview

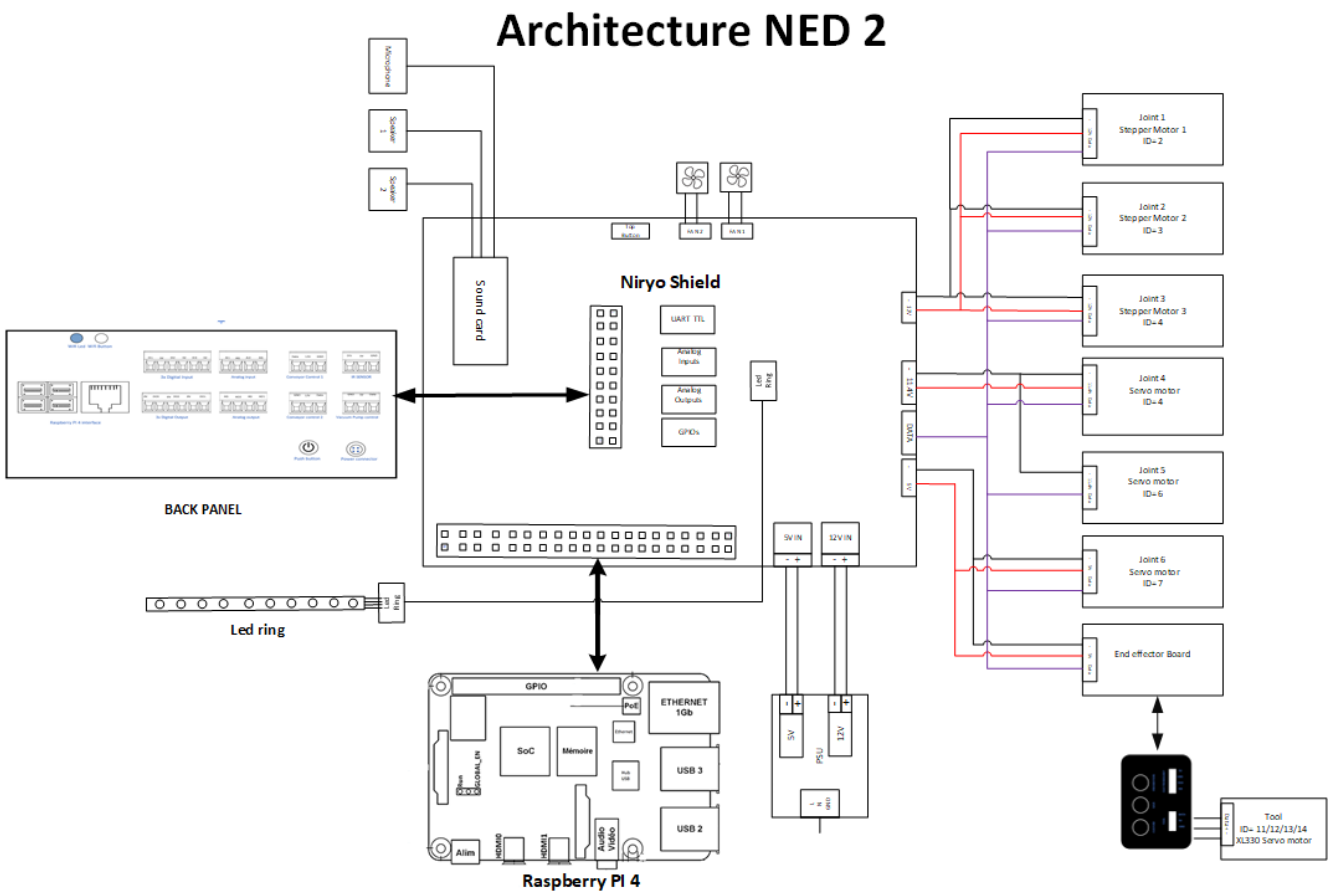
Site web du wiki généré : https://www.zetamap.fr/niryo_robot_project/wiki/

Outil de génération de documentation pour Python : https://github.com/twisted/pydoctor/

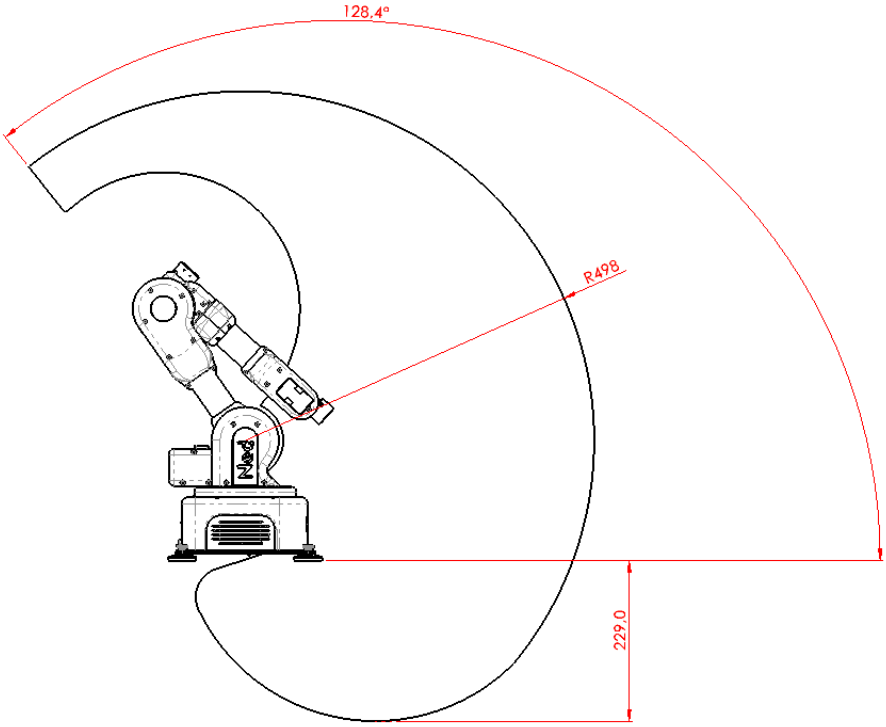
Annexe 1 : Gestion de projet Trello

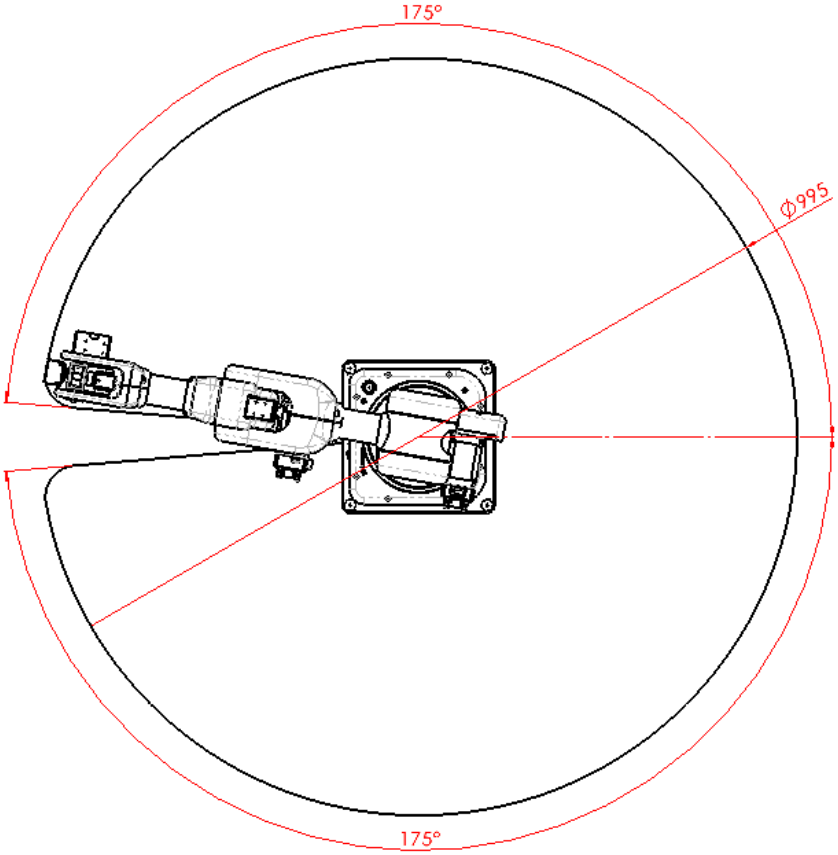


Annexe 2 : Schéma électrique Niryo Ned 2

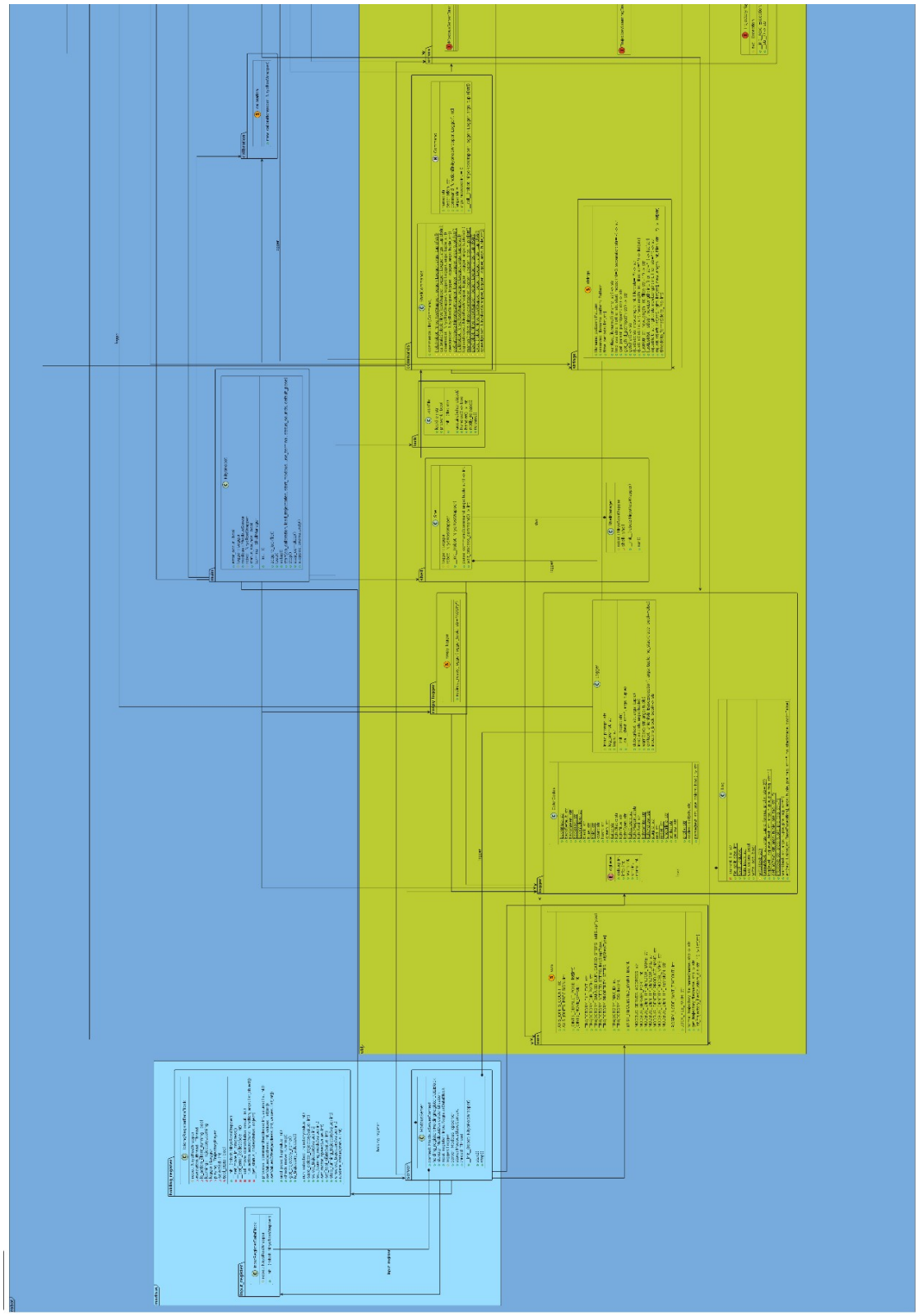


Annexe 3 : Espace de travail Niryo Ned 2





Annexe 4 : Extrait du diagramme de classe



Annexe 5 : Premier programme

```python
from util.logger import Logger
logger = Logger(__file__[__file__.rfind('/')+1 or
__file__.rfind('\\')+1:__file__.rfind('.')])

logger.info("Importing libraries")
import rospy
from niryo_robot_python_ros_wrapper.ros_wrapper import
NiryoRosWrapper

from trajectory.step.goal import GoalReport, GoalFailure
from util.colors import Colors

logger.info("Starting ROSPy")
rospy.init_node('niryo_test_FQC_ros_wrapper')
logger.info("Connecting to robot")
robot = NiryoRosWrapper()


def new_calibration(robot: NiryoRosWrapper):
  robot.clear_collision_detected()
  robot.request_new_calibration()
  robot.calibrate_auto()
  robot.update_tool()
  rospy.sleep(0.1)
  robot.grasp_with_tool()
  rospy.sleep(0.1)
  robot.release_with_tool()

IS_LEARNING = False
LEARN_STEPS = []
LEARNED = {}

def activate_learning():
  global IS_LEARNING, LEARN_STEPS
  if IS_LEARNING:
    logger.warn("Robot already learning")
    return
  if len(LEARN_STEPS) != 0:
    logger.err("Learned trajectory must be saved or cleaned before
learning")
    return
  add_step()
  add_tool_state(False)
  IS_LEARNING = True
  logger.info("Robot is learning ...")

def deactivate_learning():
  global IS_LEARNING
  if not IS_LEARNING:
    logger.warn("Robot is not learning")
    return
  IS_LEARNING = False
  logger.info("Robot has stopped learning ...")

def add_step():
  global LEARN_STEPS
  LEARN_STEPS.append({"pos": robot.get_joints()})

def add_tool_state(enabled: bool):
  global LEARN_STEPS
  if enabled:
    robot.grasp_with_tool()
```

```python
def getchar():
  # Returns a single character from standard input
  import os
  ch = ''
  if os.name == 'nt': # how it works on windows
    import msvcrt
    ch = msvcrt.getch()
  else:
    import tty, termios, sys
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    try:
      tty.setraw(sys.stdin.fileno())
      ch = sys.stdin.read(1)
    finally:
      termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
  if ord(ch) == 3: raise KeyboardInterrupt # handle ctrl+C
  return ch

MOVE_OFFSET = 0.1
tool_state = False
new_calibration(robot)
try:
  robot.sound.play("ready.wav")
  activate_learning()
  ### Put the main code bellow
  while True:
    key = getchar()
    logger.info(f"Received {key!r}")

    last_pos = robot.get_joints()
    if key == 'a':
      last_pos[0] += MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'q':
      last_pos[0] -= MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'z':
      last_pos[1] += MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 's':
      last_pos[1] -= MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'e':
      last_pos[2] += MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'd':
      last_pos[2] -= MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'r':
      last_pos[3] += MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'f':
      last_pos[3] -= MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 't':
      last_pos[4] += MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'g':
      last_pos[4] -= MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'y':
```

```
    LEARN_STEPS.append({"tool": True})
  else:
    robot.release_with_tool()
    LEARN_STEPS.append({"tool": False})

def save_learned(name: str):
  global IS_LEARNING, LEARN_STEPS, LEARNED
  if IS_LEARNING:
    logger.err("Stop learning first")
    return
  if len(LEARN_STEPS) == 0:
    logger.err("No steps to save")
    return
  LEARNED.update({name: LEARN_STEPS.copy()})
  clear_learned()

def clear_learned():
  global LEARN_STEPS
  LEARN_STEPS.clear()

def play_learned(name: str):
  global IS_LEARNING, LEARNED
  if IS_LEARNING:
    logger.err("Cannot play when learning")
    return
  steps = LEARNED.get(name, None)
  if not steps:
    logger.err(f"No learned trajectory with name {name!r}")
    return
  for s in steps:
    step = s.get("tool", None)
    if step is not None:
      if step: robot.grasp_with_tool()
      else: robot.release_with_tool()
      continue
    step = s.get("pos", None)
    if step is not None:
      try:GoalReport(robot.move_joints, "Move joints",
*step).raise_if_error()
      except GoalFailure as e:
        logger.err(e, no_stacktrace=True)
        return
      continue
    logger.err("Unknown step type: " + ", ".join(list(s.keys())))
```

```
      last_pos[5] += MOVE_OFFSET
      robot.move_joints(*last_pos)
    elif key == 'h':
      last_pos[5] -= MOVE_OFFSET
      robot.move_joints(*last_pos)

    elif key == ' ': # add step
      add_step()
      robot.led_ring.flashing(Colors.cyan, 0.5, 2, True)
    elif key == '\t': # invert tool state
      tool_state = not tool_state
      add_tool_state(tool_state)
    elif key == '\r': # finish learning
      deactivate_learning()
      save_learned("test")
      break

  if robot.custom_button.wait_for_any_action() != 100:
    robot.clear_collision_detected()

    play_learned("test")


except BaseException as e:
  if (type(e) not in (SystemExit, KeyboardInterrupt) and
      "returned no response" not in '. '.join(str(i) for i in
e.args).lower()):
    logger.err(e)
finally:
  rospy.core._shutdown_flag = False
  try: robot.sound.play("disconnected.wav")
  except BaseException as e: logger.err(e, no_stacktrace=True)
  rospy.core._shutdown_flag = True
```

Annexe 6 : Code du conteneur de trajectoire

```python
from .step.type import *
from .step.value import *
from .controller import TrajectoryStep, PropertyStep, TrajectoryController
from util.vars import TRAJECTORY_MAX_ID, TRAJECTORY_IDS

class Trajectory(TrajectoryController):
  @staticmethod
  def clear_all_ids():
    TRAJECTORY_IDS.clear()

  def new_id(self) -> TrajectoryController:
    size = len(TRAJECTORY_IDS)
    if size > TRAJECTORY_MAX_ID: raise ValueError("cannot assign new id, limit reached")
    self.set_id(0 if size == 0 else max(TRAJECTORY_IDS)+1)
    return self

  def set_property(self, step: PropertyStep) -> TrajectoryController:
    """Intercept .set_property() to save ids and avoid to duplicate it"""
    if step.type == StepTypes.id.value:
      new = step.get_value()
      if new in TRAJECTORY_IDS: raise ValueError(f"duplicated trajectory id {new!r}, use .new_id()")
      old = self.get_id()
      if old is not None: TRAJECTORY_IDS.remove(old)
      TRAJECTORY_IDS.append(new)
    return super().set_property(step)

  def set_start_state(self, tool: bool, joints: 'tuple[float]') -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.start.value, StepValue((StepValue(tool), StepValue(joints)))))
    return self
  def get_start_state(self) -> 'tuple[bool, tuple[float]]':
    result = self.get_property(StepTypes.start.value)
    return None if result is None else [v.value if isinstance(v, StepValue) else v for v in result.get_value()]

  def set_name(self, name: str) -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.label.value, StepValue(name)))
    return self
  def get_name(self) -> str:
    result = self.get_property(StepTypes.label.value)
    return None if result is None else result.get_value()

  def set_id(self, id: int) -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.id.value, StepValue(id)))
    return self
  def get_id(self) -> int:
    result = self.get_property(StepTypes.id.value)
    return None if result is None else result.get_value()

  def set_filename(self, filename: str) -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.file.value, StepValue(filename)))
    return self
  def get_filename(self) -> str:
    result = self.get_property(StepTypes.file.value)
    return None if result is None else result.get_value()

  def add_joints(self, joints: 'tuple[float]') -> 'Trajectory':
    return self.add_step(TrajectoryStep(StepTypes.move.value, StepValue(joints)))

  def add_tool_state(self, state: bool) -> 'Trajectory':
    return self.add_step(TrajectoryStep(StepTypes.tool.value, StepValue(state)))

  def add_motor_speed(self, percent: int) -> 'Trajectory':
    return self.add_step(TrajectoryStep(StepTypes.acceleration.value, StepValue(percent)))
```

Annexe 7 : Programme de tests unitaires

```
# !/usr/bin/env python3

# Test FQC V1.4

import json
import subprocess
from datetime import datetime

import numpy as np
import psutil
import requests
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper, NiryoRosWrapperException
from niryo_robot_python_ros_wrapper.ros_wrapper_enums import ButtonAction

import rospy
from niryo_robot_database.srv import SetSettings
from niryo_robot_rpi.srv import LedBlinker, LedBlinkerRequest
from niryo_robot_rpi.srv import ScanI2CBus
from std_msgs.msg import String, Int32

# This is the variable to modify for the demo program. FULL is here to define the default test (here demo)
LOOPS = 1
CALIBRATION_LOOPS = 1
SPIRAL_LOOPS = 1
HIGH_SPEED_LOOP = 10
NB_HOURS = 5
FULL = 0

SPEED = 100  # %
ACCELERATION = 100  # %

USE_BUTTON = False
VOLUME = 100
USE_VOCAL = False

WHITE = [255, 255, 255]
GREEN = [50, 255, 0]
BLACK = [0, 0, 0]
BLUE = [15, 50, 255]
PURPLE = [153, 51, 153]
PINK = [255, 0, 255]
RED = [255, 0, 0]
YELLOW = [255, 255, 0]

j_limit_1m, j_limit_1M, j_limit_2m, j_limit_2M = np.deg2rad(-164), np.deg2rad(164), np.deg2rad(-90), np.deg2rad(31)
j_limit_3m, j_limit_3M, j_limit_4m, j_limit_4M = np.deg2rad(-74), np.deg2rad(86), np.deg2rad(-116), np.deg2rad(116)
j_limit_5m, j_limit_5M, j_limit_6m, j_limit_6M = np.deg2rad(-105), np.deg2rad(105), np.deg2rad(-144), np.deg2rad(144)

def almost_equal_array(a, b, decimal=1):
    try:
        np.testing.assert_almost_equal(a, b, decimal)
        return True
    except AssertionError:
        return False

class TestStatus(object):
    FAILURE = -1
    NONE = 0
    SUCCESS = 1

class TestFailure(Exception):
    pass
```

```
class TestReport(object):  # We define the report to send to RFM

    def __init__(self, header):
        self._header = header
        self._report = ""
        self._report_dict = {}

    def __str__(self):
        return self._report

    def append(self, message):
        new_line = "[{}] - {} - {}.".format(self._header, datetime.now(), message)
        rospy.loginfo('\033[96;24;23m ' + new_line + ' \033[0m')
        self._report += new_line

    def execute(self, function, prefix, args=[]):  # Execute a function and create a report of the latter
        try:
            reply = function(*args)
            status, message = reply if reply else (0,"")
        except Exception as e:
            self.append("{}{} - {}".format(prefix, " failed" if prefix else "Failed", str(e)))
            raise TestFailure(e)
        else:
            if status >= 0:
                success_message = " succeed" if prefix else "Succeed"
                self.append("{}{} - {} - {}".format(prefix, success_message, status, message))
                return status
            else:
                success_message = " failed" if prefix else "Failed"
                self.append("{}{} - {} - {}".format(prefix, success_message, status, message))
                raise TestFailure(message)

    @property
    def report(self):
        return self._report

class TestStep(object):  # This is the part who execute each program

    def __init__(self, function, name, critical=False):
        self.__function = function
        self.__name = name
        self.__report = TestReport(name)
        self.__critical = critical

        self.__status = TestStatus.NONE

    def run(self, *args, **kwargs):  # run each function
        self.__status = TestStatus.NONE

        try:
            self.__function(self.__report)
        except TestFailure as e:
            error_message = "[Error] [{}] - {}".format(self.__name, str(e))
            self.__report.append(error_message)
        except Exception as e:
            error_message = "[Error] [{}] - {}".format(self.__name, str(e))
            self.__report.append(error_message)
        else:
            self.__status = TestStatus.SUCCESS
            return True

        self.__status = TestStatus.FAILURE
        if self.__critical:
            raise TestFailure(error_message)
        return False

    def get_report(self):
```

```
        return {"name": self.__name, "status": self.__status, "report": str(self.__report)}

class TestProduction:  # Here we create the program who contain each function that we want

    def __init__(self):
        self.__functions = TestFunctions()
        self.__success = False

        if FULL == 1:  # For the short test (freemotion and button)
            self.__sub_tests = [
                TestStep(self.__functions.name, "Program name", critical=True),
                TestStep(self.__functions.test_cloud_connection, "Test robot connection", critical=True),
                TestStep(self.__functions.test_robot_status, "Test robot status", critical=True),
                TestStep(self.__functions.test_calibration, "Test calibration", critical=True),
                TestStep(self.__functions.test_sound, "Test haut parleurs", critical=True),
                TestStep(self.__functions.test_led_ring, "Test led ring", critical=True),
                TestStep(self.__functions.test_joint_limits, "Test joint limits", critical=True),
                TestStep(self.__functions.test_spiral, "Test spiral", critical=True),
                TestStep(self.__functions.test_fun_poses, "Test poses", critical=True),
                TestStep(self.__functions.test_pick_and_place, "Test pick and place", critical=True),
                TestStep(self.__functions.test_high_speed, "Test high speed", critical=True),
                TestStep(self.__functions.end_test, "Test final check", critical=True)
            ]
        if FULL == 2:  # For the long test (custom and button)
            self.__sub_tests = [
                TestStep(self.__functions.name, "Program name", critical=True),
                TestStep(self.__functions.test_cloud_connection, "Test robot connection", critical=True),
                TestStep(self.__functions.test_robot_status, "Test robot status", critical=True),
                TestStep(self.__functions.test_calibration, "Test calibration", critical=True),
                TestStep(self.__functions.test_spiral, "Test spiral", critical=True),
                TestStep(self.__functions.test_long, "Long test", critical=True),
                TestStep(self.__functions.end_test, "Test final check", critical=True)
            ]
        if FULL == 0:  # For the demo (only button) and also default mode
            self.__sub_tests = [
                TestStep(self.__functions.name, "Program name", critical=True),
                TestStep(self.__functions.test_calibration, "Test calibration", critical=True),
                TestStep(self.__functions.test_spiral, "Test spiral", critical=True),
                TestStep(self.__functions.test_joint_limits, "Test joint limits", critical=True),
                TestStep(self.__functions.test_fun_poses, "Test poses", critical=True),
                TestStep(self.__functions.test_pick_and_place, "Test pick and place", critical=True),
                TestStep(self.__functions.end_test, "Test final check", critical=True)
            ]
        if FULL == 3:  # Expedition test
            self.__sub_tests = [
                TestStep(self.__functions.name, "Program name", critical=True),
                TestStep(self.__functions.test_cloud_connection, "Test robot connection", critical=True),
                TestStep(self.__functions.test_robot_status, "Test robot status", critical=True),
                TestStep(self.__functions.test_calibration, "Test calibration", critical=True),
                #TestStep(self.__functions.test_sound, "Test haut parleurs", critical=True),
                TestStep(self.__functions.test_led_ring, "Test led ring", critical=True),
                TestStep(self.__functions.test_joint_limits, "Test joint limits", critical=True),
                TestStep(self.__functions.test_spiral, "Test spiral", critical=True),
                TestStep(self.__functions.test_fun_poses, "Test poses", critical=True),
                TestStep(self.__functions.test_pick_and_place, "Test pick and place", critical=True),
                TestStep(self.__functions.end_test, "Test final check", critical=True)
            ]

    def run(self):  # run the whole program
        rospy.sleep(1)
        self.__functions.led_stop()

        for test in self.__sub_tests:
            try:
                test.run()

            except TestFailure:  # What's append if the test fail
                self.__sub_tests.append(TestStep(self.__functions.test_robot_status, "Test robot status", critical=True))
```

```
                # self.__sub_tests[-1].run()
                self.__functions.reset()
                self.__success = False
                self.__functions.led_error()

            else:
                self.__success = True

        return self.__success

    def get_report(self):
        return {"details": [test.get_report() for test in self.__sub_tests], "success": self.__success}

    def print_report(self):
        print(json.dumps(self.get_report(), indent=4, sort_keys=True))

    def send_report(self):
        new_report_publisher = rospy.Publisher('/niryo_robot_reports/test_report', String, queue_size=10)
        rospy.sleep(0.5)

        # Wait for the publisher initialization
        start_time = rospy.Time.now()
        while not rospy.is_shutdown() and new_report_publisher.get_num_connections() == 0:
            if (rospy.Time.now() - start_time).to_sec() > 1:
                rospy.logerr('Unable to publish the new report')
                return False
            rospy.sleep(0.1)

        new_report_publisher.publish(json.dumps(self.get_report()))
        rospy.sleep(1)

        rospy.logdebug('test report published')

        return True

class TestFunctions(object):  # definition of each function (some are unused)

    def __init__(self):
        rospy.sleep(2)
        self.__robot = robot
        self.__hardware_version = self.__robot.get_hardware_version()

        self.__robot.set_arm_max_velocity(SPEED)
        self.__robot.set_arm_max_acceleration(ACCELERATION)
        self.__set_led_state_service = rospy.ServiceProxy('/niryo_robot_rpi/set_led_custom_blinker', LedBlinker)
        self.led_stop()

    def reset(self):
        self.__robot.set_arm_max_velocity(100)
        self.__robot.set_arm_max_acceleration(100)

    def led_stop(self):
        self.__robot.led_ring.turn_off()

    def led_error(self, duration=360):
        if self.__hardware_version in ['ned', 'one'] and not self.__robot.get_simulation_mode():
            self.__set_led_state_service(False, 0, 0, 0)

    def say(self, text, prio=0):  # prio is here to speak even if USE_VOCAL = False
        if (USE_VOCAL or prio == 1) and self.__hardware_version in ['ned2']:
            try:
                self.__robot.sound.say(text, 1)
            except Exception:
                rospy.loginfo("No internet connection, robot can't speak")
        subprocess.run('rm -rf /home/niryo/niryo_robot_saved_files/niryo_robot_user_sounds/last_text_to_speech.mp3',
                       shell=True,
                       stdout=subprocess.DEVNULL,
                       stderr=subprocess.DEVNULL)
```

```
    def name(self, report):
        if FULL != 0:
            report.execute(self.__robot.sound.set_volume, "Set volume", [VOLUME])

        name = ["Blablacar", "Debut du test court FQC", "Debut du test long FQC",
                "Bouzazou magique"]  # expedition
        self.say(name[FULL], 1)

    def ip_say(self, report):
        self.__robotblablacar.led_ring.solid(PURPLE)
        ip_eth = "Adresse ip ethernet : " + psutil.net_if_addrs()['eth0'][0].address
        ip_wifi = "Adresse ip wifi : " + psutil.net_if_addrs()['wlan0'][0].address
        while True:  # repeat fonction
            try:
                value = report.execute(self.wait_custom_button_press, "Wait button press to validate", ret=True)

                if value == 2:  # short press
                    self.say(ip_wifi)
                    self.say(ip_eth)
                if value == 1:  # long press
                    break
            except Exception as e:
                report.append(e)
                raise TestFailure(e)
        report.append("Voice step successfully pass")

    def test_cloud_connection(self, report):

        rasp_id = self.__robot.database.get_setting('rasp_id')
        apiKey = self.__robot.database.get_setting('api_key')
        try:
            response = requests.get('https://api.niryo.com/test-reports/api/v1/test-reports/ping',
                                    headers={
                                        'accept': 'application/json', 'identifier': rasp_id, 'apiKey': apiKey
                                    })
            if response.status_code >= 400:
                report.append("error {}".format(response.text))
                self.say("Erreur connexion RFM", 1)
                raise TestFailure("Erreur connexion RFM")
            else:
                report.append("Service test_reports successfully reached")
                self.say("Connexion au serveur RFM rai hussi", 1)  # reussi
        except Exception as e:
            report.append(e)
            report.append("No internet connection")
            raise TestFailure(e)

    def  test_robot_status(self, report):
        try:
            hardware_status = self.__robot.get_hardware_status()
        except rospy.exceptions.ROSException as e:
            report.append(str(e))
            raise TestFailure(e)

        if hardware_status.error_message:
            message = "Hardware status Error - {}".format(hardware_status.error_message)
            report.append(message)
            raise TestFailure(message)

        if any(hardware_status.hardware_errors):
            message = "Hardware status Motor Error - {}".format(hardware_status.hardware_errors_message)
            report.append(message)
            raise TestFailure(message)

        if hardware_status.rpi_temperature > 70:
            message = "Rpi overheating"
            report.append(message)
```

```
            raise TestFailure(message)
        report.append("Hardware status ok")

        try:
            robot_status = self.__robot.get_robot_status()
        except rospy.exceptions.ROSException as e:
            report.append(str(e))
            raise TestFailure(e)

        if robot_status.robot_status < 0:
            message = "Robot status - {} - {}".format(robot_status.robot_status_str, robot_status.robot_message)
            report.append(message)
            raise TestFailure(message)
        report.append("Robot status ok")

        def test_i2c():
            if self.__robot.get_simulation_mode():
                report.append("I2C Bus - Skipped in simulation mode")
                return

            try:
                rospy.wait_for_service("/niryo_robot_rpi/scan_i2c_bus", timeout=0.2)
                resp = rospy.ServiceProxy("/niryo_robot_rpi/scan_i2c_bus", ScanI2CBus).call()

                if not resp.is_ok:
                    message = "I2C Bus - Missing components: {}".format(resp.missing)
                    report.append(message)
                    raise TestFailure(message)

            except (rospy.ROSException, rospy.ServiceException) as e:
                report.append(str(e))
                raise TestFailure(e)

        if self.__hardware_version in ['ned2']:
            test_i2c()

    def test_calibration(self, report):
        for i in range(CALIBRATION_LOOPS):

            self.__robot.request_new_calibration()
            rospy.sleep(0.1)
            report.execute(self.__robot.calibrate_auto, "Calibration")
            self.__robot.set_learning_mode(False)
            rospy.sleep(0.1)
            report.execute(self.move_and_compare, "Move after calibration", args=[6 * [0], 1])

            if self.__hardware_version in ['ned2']:
                if i == 0:
                    self.__robot.led_ring.flashing(BLUE)
                else:
                    self.__robot.led_ring.flashing(PURPLE)

                self.say("Appuyez sur SAVE pour valider la position du elbo a 90 degre")

                report.execute(self.wait_save_button_press, "Wait save button press to validate")

            self.__robot.move_to_sleep_pose()

    def test_led_ring(self, report):
        if self.__hardware_version not in ['ned2']:
            report.append("Led ring test - Skipped on {}".format(self.__hardware_version))
            return

        self.__robot.led_ring.solid(WHITE)

        self.say("Premier test du ruban led")
        report.append("Led ring color set to WHITE")
        self.say("Appuyer sur custom pour valider le LED RING Blanc")
```

```
        report.execute(self.wait_custom_button_press, "Wait custom button press to continue")

        self.__robot.led_ring.rainbow_cycle()
        report.append("Led ring color set to RAINBOW")
        self.say("Appuyer sur CUSTOM pour valider le LED RING multicolor tournant")
        self.say("Validez le test")
        report.execute(self.wait_custom_button_press, "Wait custom button press to validate")

    def test_sound(self, report):
        if self.__hardware_version not in ['ned2']:
            report.append("Sound test - Skipped on {}".format(self.__hardware_version))
            return

        self.__robot.led_ring.solid(PURPLE)

        self.say("Test des haut parleurs")

        def play_channel(channel):
            subprocess.run((f'ffplay -nodisp -autoexit -af "pan=stereo|c{channel}=c0+c1" '
                            f'/home/niryo/catkin_ws/src/niryo_robot_sound/niryo_robot_state_sounds/reboot.wav'),
                           shell=True,
                           stdout=subprocess.DEVNULL,
                           stderr=subprocess.DEVNULL)

        report.execute(self.__robot.sound.set_volume, "Set volume", [int(VOLUME / 2)])
        report.append("Volume set to {}%".format(int(VOLUME / 2)))
        self.say("Volume haut parleur gauche 50 pour 100")

        play_channel(1)

        report.execute(self.__robot.sound.set_volume, "Set volume", [VOLUME])
        report.append("Volume set to {}%".format(VOLUME))
        self.say("Volume haut parleur gauche 100 pour 100")

        play_channel(1)

        report.execute(self.__robot.sound.set_volume, "Set volume", [int(VOLUME / 2)])
        report.append("Volume set to {}%".format(int(VOLUME / 2)))
        self.say("Volume haut parleur droit 50 pour 100")

        play_channel(0)

        report.execute(self.__robot.sound.set_volume, "Set volume", [VOLUME])
        report.appenblablacard("Volume set to {}%".format(VOLUME))
        self.say("Volume haut parleur droit 100 pour 100")

        play_channel(0)

        self.say("Appuyez sur CUSTOM pour validez le test des haut parleurs")
        report.execute(self.wait_custom_button_press, "Wait custom button press to validate")

    def test_freedrive(self, report):
        if self.__hardware_version not in ['ned2']:
            report.append("Freemotion test - Skipped on {}".format(self.__hardware_version))
            return

        def wait_learning_mode(value):
            start_time = rospy.Time.now()
            while self.__robot.get_learning_mode() == value:
                if (rospy.Time.now() - start_time).to_sec() > 20:
                    return -1, "Timeout: no detected".format("learning mode" if value else "torque on")
                rospy.sleep(0.1)

            return 1, "Learning mode".format("enabled" if value else "disabled")

        self.say("Test de free motion")

        joint_limit = self.__robot.get_axis_limits()[1]['joint_limits']
```

```
        self.__robot.led_ring.solid(GREEN)
        report.append("Led ring color set to GREEN")

        report.append("Wait learning mode")
        report.execute(wait_learning_mode, "Wait learning mode", [True])

        for i in range(0, 30, 6):
            for j in range(2):
                self.__robot.led_ring.set_led_color(i + j, BLACK)

        report.append("Wait joint1 minimum limit")
        start_time = rospy.Time.now()
        while not self.__robot.get_joints()[0] < joint_limit['joint_1']['min'] + 0.2:
            if (rospy.Time.now() - start_time).to_sec() > 20:
                report.append("Joint1 minimum limit not reached")
                break
        else:
            self.say("Limite validee")
            report.append("Joint1 minimum limit reached")

        for i in range(2, 30, 6):
            for j in range(2):
                self.__robot.led_ring.set_led_color(i + j, BLACK)

        report.append("Wait joint1 minimum limit")
        start_time = rospy.Time.now()
        while not self.__robot.get_joints()[0] > joint_limit['joint_1']['max'] - 0.2:
            if (rospy.Time.now() - start_time).to_sec() > 20:
                report.append("Joint1 maximum limit not reached")
                break
        else:
            self.say("Limite validee")
            report.append("Joint1 maximum limit reached")

        for i in range(4, 30, 6):
            for j in range(2):
                self.__robot.led_ring.set_led_color(i + j, BLACK)

        rospy.sleep(1)
        self.__robot.led_ring.flashing(GREEN)

        report.execute(wait_learning_mode, "Wait learning mode disabled", [False])
        rospy.sleep(1)

    def test_io(self, report):
        if self.__hardware_version not in ['ned2']:
            report.append("IO test - Skipped on {}".format(self.__hardware_version))
            return

        def test_digital_io_value(io_name, state):
            io_state = self.__robot.digital_read(io_name)
            if io_state != state:
                raise TestFailure("Non expected value on digital input {} - Actual {} - Target {}".format(
                    io_name, io_state, state))
            return 1, "Success"

        def test_analog_io_value(io_name, value, error=0.3):
            io_state = self.__robot.analog_read(io_name)
            if not (value - error <= io_state <= value + error):
                raise TestFailure("Non expected value on digital input {} - Actual {} - Target {}".format(
                    io_name, io_state, value))
            return 1, "Success"

        self.__robot.led_ring.solid(PINK)

        # Test digital ios
        dio = self.__robot.get_digital_io_state()
```

```
        for do in dio.digital_output:
            report.execute(self.__robot.digital_write, 'Set digital output {}'.format(do.name), [do.name, True])
        for di in dio.digital_input:
            report.execute(test_digital_io_value, 'Test digital input {} is High'.format(di.name), [di.name, True])

        # Test analog ios
        aio = self.__robot.get_analog_io_state()
        for ao in aio.analog_outputs:
            report.execute(self.__robot.digital_write, 'Set analog output {} to 5V'.format(ao.name), [ao.name, 5.0])
        for ai in aio.analog_inputs:
            report.execute(test_analog_io_value, 'Test analog input {} is 5V'.format(ai.name), [ai.name, 5.0])

        self.__robot.led_ring.flashing(PINK)
        self.wait_custom_button_press()

        for do in dio.digital_output:
            report.execute(self.__robot.digital_write, 'Unset digital output {}'.format(do.name), [do.name, False])
        for di in dio.digital_input:
            report.execute(test_digital_io_value, 'Test digital input {} is Low'.format(di.name), [di.name, False])
        for ao in aio.analog_outputs:
            report.execute(self.__robot.digital_write, 'Set analog output {} to 0V'.format(ao.name), [ao.name, 0.0])
        for ai in aio.analog_inputs:
            report.execute(test_analog_io_value, 'Test analog input {} is 0V'.format(ai.name), [ai.name, 0.0])

    def test_joint_limits(self, report):
        if self.__hardware_version in ['ned2']:
            self.__robot.led_ring.rainbow_cycle()
            self.say("Test des limites des joints")

        self.__robot.set_learning_mode(False)
        rospy.sleep(1)

        default_joint_pose = 6 * [0.0]

        first_target, second_target, third_target, last_target = 6 * [0], 6 * [0], 6 * [0], 6 * [0]

        first_target[0], first_target[3], first_target[4], first_target[
            5] = j_limit_1m, j_limit_4m, j_limit_5m, j_limit_6m

        second_target[1], second_target[2] = j_limit_2M, j_limit_3m

        third_target[0], third_target[3], third_target[4], third_target[
            5] = j_limit_1M, j_limit_4M, j_limit_5M, j_limit_6M

        last_target[2], last_target[4] = j_limit_3M, j_limit_5m

        poses = [(default_joint_pose, 1, 3), (first_target, 1, 4), (default_joint_pose, 1, 4), (second_target, 1, 3),
                 (default_joint_pose, 1, 3), (third_target, 1, 4), (last_target, 1, 4)]

        for loop_index in range(LOOPS):
            for position_index, step in enumerate(poses):
                joint_position, precision, duration = step
                report.execute(self.move_and_compare_without_moveit,
                               "Move number {}.{}".format(loop_index, position_index),args=[joint_position, precision,
duration])

    def test_spiral(self, report):
        if self.__hardware_version in ['ned2']:
            self.__robot.led_ring.rainbow_cycle()
            self.say("Test des spirales")

        for loop_index in range(SPIRAL_LOOPS):
            report.execute(self.__robot.move_pose,
                           "Loop {} - Move to spiral center".format(loop_index), [0.3, 0, 0.2, 0, 1.57, 0])
            report.execute(self.__robot.move_spiral, "Loop {} - Execute spiral".format(loop_index), [0.15, 5, 216, 3])

    def test_fun_poses(self, report):
```

```
        if self.__hardware_version in ['ned2']:
            self.__robot.led_ring.rainbow_cycle()
            self.say("Test de divers movements")

        waypoints = [[0.16, 0.00, -0.75, -0.56, 0.60, -2.26], [2.25, -0.25, -0.90, 1.54, -1.70, 1.70],
                     [1.40, 0.35, -0.34, -1.24, -1.23, -0.10], [0.00, 0.60, 0.46, -1.55, -0.15,
                                                                2.50], [-1.0, 0.00, -1.00, -1.70, -1.35, -0.14]]

        for loop_index in range(LOOPS):
            for wayoint_index, wayoint in enumerate(waypoints):
                report.execute(self.move_and_compare_without_moveit,
                               "Loop {}.{} - Fun move".format(loop_index, wayoint_index),
                               args=[wayoint, 1, 4])

    def test_pick_and_place(self, report):
        report.execute(self.move_and_compare, "Move to 0.0", args=[6 * [0], 1])

        if self.__hardware_version in ['ned2']:
            self.say("Mettre le grippeur et appuyer sur CUSTOM")
            self.__robot.led_ring.flashing(YELLOW)
            self.wait_custom_button_press()
            self.__robot.led_ring.solid(YELLOW)
            self.say("Test de pick and place")

        report.execute(self.__robot.update_tool, "Scan tool")
        report.append("Detected tool: {}".format(self.__robot.get_current_tool_id()))

        self.__robot.enable_tcp(True)

        z_offset = 0.15 if self.__robot.get_current_tool_id() <= 0 else 0.02
        sleep_pose = [0.3, 0, 0.3, 0, 1.57, 0]
        home_pose = [0.3, 0, 0.3, 0, 0, 0]
        pick_1 = [0, 0.2, z_offset, 0, 1.57, 0]
        pick_2 = [0, -0.2, z_offset, 0, 1.57, 0]
        place_1 = [0.15, 0, z_offset, 0, 1.57, 0]
        place_2 = [0.22, 0, z_offset, 0, 1.57, 0]

        report.execute(self.__robot.move_pose, "Move to sleep pose", sleep_pose)

        report.execute(self.__robot.pick_from_pose, "Pick 1st piece", pick_1)
        report.execute(self.__robot.place_from_pose, "Place 1st piece", place_1)

        report.execute(self.__robot.move_pose, "Move to sleep pose", sleep_pose)

        report.execute(self.__robot.pick_from_pose, "Pick 2nd piece", pick_2)
        report.execute(self.__robot.place_from_pose, "Place 2nd piece", place_2)

        report.execute(self.__robot.move_pose, "Move to sleep pose", sleep_pose)

        report.execute(self.__robot.pick_from_pose, "Pick 1st piece from center", place_1)
        pick_1[5] = 1.57
        report.execute(self.__robot.place_from_pose, "Replace 1st piece", pick_1)

        report.execute(self.__robot.move_pose, "Move to sleep pose", sleep_pose)

        report.execute(self.__robot.pick_from_pose, "Pick 2nd piece from center", place_2)
        pick_2[5] = -1.57
        report.execute(self.__robot.place_from_pose, "Replace 2nd piece", pick_2)

        try:
            report.execute(self.__robot.grasp_with_tool, "Close gripper")
        except Exception:
            report.append("No tool connected")

        report.execute(self.__robot.move_pose, "Move to home pose", home_pose)

        self.__robot.enable_tcp(False)
```

```
    def test_high_speed(self, report):

        self.__robot.set_arm_max_velocity(200)
        self.__robot.set_arm_max_acceleration(100)

        default_joint_pose = 6 * [0.0]

        first_joint_min, first_joint_max, second_joint_min, second_joint_max = 6 * [0], 6 * [0], 6 * [0], 6 * [0]
        third_joint_min, third_joint_max, fourth_joint_min, fourth_joint_max = 6 * [0], 6 * [0], 6 * [0], 6 * [0]
        fifth_joint_min, fifth_joint_max, sixth_joint_min, sixth_joint_max = 6 * [0], 6 * [0], 6 * [0], 6 * [0]

        first_joint_min[0], first_joint_min[1], first_joint_min[2] = j_limit_1m, j_limit_2m, j_limit_3M
        first_joint_max[0], first_joint_max[1], first_joint_max[2] = j_limit_1M, j_limit_2m, j_limit_3M

        second_joint_min[1], second_joint_min[2] = j_limit_2m, j_limit_3M
        second_joint_max[1], second_joint_max[2] = j_limit_2M, j_limit_3M

        third_joint_min[1], third_joint_min[2] = j_limit_2M, j_limit_3m
        third_joint_max[1], third_joint_max[2] = j_limit_2M, j_limit_3M

        fourth_joint_min[3], fourth_joint_min[4] = j_limit_4m, np.deg2rad(-90)
        fourth_joint_max[3], fourth_joint_max[4] = j_limit_4M, np.deg2rad(-90)

        fifth_joint_min[4] = j_limit_5m
        fifth_joint_max[4] = j_limit_5M

        sixth_joint_min[5] = j_limit_6m
        sixth_joint_max[5] = j_limit_6M

        poses = [
            first_joint_min,
            first_joint_max,
            second_joint_min,
            second_joint_max,
            third_joint_min,
            third_joint_max,
            fourth_joint_min,
            fourth_joint_max,
            fifth_joint_min,
            fifth_joint_max,
            sixth_joint_min,
            sixth_joint_max
        ]

        c = [-1, 0, 0, 2, 1, 0]  # correctif for 10 loop and 1min per axis :  c = [-1, 0, 0, 2, 1, 0]

        self.say("Mettre la masse et appuyer sur CUSTOM")
        report.execute(self.wait_custom_button_press, "Wait custom button press to validate")

        for position_index in range(int(len(poses) / 2)):

            if position_index == 3:  # limit of the axe 4
                self.__robot.set_arm_max_velocity(150)  # to avoid problem
            if position_index == 4:  # 150% speed is enough
                self.__robot.set_arm_max_velocity(200)  #
            if position_index == 5:
                report.execute(self.move_and_compare, "Change payload", args=[default_joint_pose])
                self.say("Mettre la masse des porter et appuyer sur CUSTOM")  # masse deportee
                report.execute(self.wait_custom_button_press, "Wait custom button press to validate")

            for loop_index in range(HIGH_SPEED_LOOP + c[position_index]):
                if position_index == 3 and loop_index == 6:  # change position of the Payload
                    fourth_joint_min[4] = np.deg2rad(90)  # at the half of the movements
                    fourth_joint_max[4] = np.deg2rad(90)  #

                joint_position_min = poses[2 * position_index]
                joint_position_max = poses[2 * position_index + 1]
                report.execute(self.move_and_compare,
```

```
                            "Move number {}.{} min ".format(position_index + 1, loop_index),
                            args=[joint_position_min])
                report.execute(self.move_and_compare,
                            "Move number {}.{} max".format(position_index + 1, loop_index),
                            args=[joint_position_max])

        report.execute(self.move_and_compare, "Move HOME", args=[default_joint_pose])
        self.say("Enlever la masse des porter et appuyer sur CUSTOM ")
        report.execute(self.wait_custom_button_press, "Wait custom button press to validate")
        self.__robot.set_arm_max_velocity(80)
        self.__robot.set_arm_max_acceleration(50)

    def test_long(self, report):

        self.__robot.set_arm_max_velocity(100)
        self.__robot.set_arm_max_acceleration(100)
        self.__robot.led_ring.solid(GREEN)
        nb_collision, nb_loop = 0, 0
        report.append("Start of the long test")

        waypoints = [[j_limit_1m, j_limit_2M, np.deg2rad(-55), j_limit_4M, j_limit_5m,
                      j_limit_6M], [j_limit_1M, j_limit_2M, j_limit_3m, j_limit_4m, j_limit_5M, j_limit_6m],
                     [j_limit_1M, j_limit_2M, j_limit_3M, j_limit_4m, j_limit_5M,
                      j_limit_6m], [j_limit_1m, j_limit_2M, j_limit_3M, j_limit_4M, j_limit_5m, j_limit_6M],
                     [j_limit_1m, j_limit_2m, j_limit_3M, j_limit_4M, j_limit_5M,
                      j_limit_6M], [j_limit_1M, j_limit_2m, j_limit_3M, j_limit_4m, j_limit_5m,
                                    j_limit_6m], [0, 0.5, -1.25, 0, 0, 0]]

        start_time = rospy.Time.now()

        while (rospy.Time.now() - start_time).to_sec() < NB_HOURS * 3600:
            nb_loop += 1
            for wayoint_index, wayoint in enumerate(waypoints):
                try:
                    self.__robot.move_joints(*wayoint)

                except NiryoRosWrapperException as e:
                    self.__robot.clear_collision_detected()
                    report.append("{}".format(str(e)))
                    nb_collision += 1
                    if nb_collision >= 5:
                        report.append("End of the test with {} loop and {} collision(s)".format(nb_loop, nb_collision))
                        raise TestFailure("Stop program : number of collision higher than 5")
                    self.__robot.wait(5)
                    if nb_collision >= 3:
                        self.__robot.led_ring.flashing(YELLOW)
        report.append("End of the test with {} loop and {} collision(s)".format(nb_loop, nb_collision))

        if nb_collision > 2:
            raise TestFailure("Number of collision higher than 2")

    def end_test(self, report):
        report.execute(self.move_and_compare, "Move to 0.0", args=[6 * [0], 1])

        if self.__hardware_version in ['ned2']:
            self.__robot.led_ring.flashing(BLUE)
            report.append("End")
            if FULL == 0:
                self.say("Fin de la demo", 1)
            if FULL == 1:
                self.say("Appuyez sur SAVE pour valider la position du elbo a 90 degrer")
                report.execute(self.wait_save_button_press, "Wait save button press to validate")
                self.say("Fin du test court FQC")
            if FULL == 2:
                self.say("Fin du test long FQC", 1)
            if FULL == 3:
                self.say("Appuyez sur SAVE pour valider la position du elbo a 90 degrer")
                report.execute(self.wait_save_button_press, "Wait save button press to validate")
```

```
                self.say("Fin du test avant expedission")
        self.__robot.led_ring.solid(BLUE)
        self.__robot.move_to_sleep_pose()
        self.__robot.set_arm_max_velocity(100)
        self.__robot.set_arm_max_acceleration(100)

    def wait_custom_button_press(self, timeout=600):
        if not USE_BUTTON:
            # raw_input('Enter to continue')
            return 1, "Press custom button step skipped"

        action = self.__robot.custom_button.wait_for_any_action(timeout=timeout)

        if action == ButtonAction.NO_ACTION:
            return -1, "Timeout: no press detected"

        return action, "Press detected"

    @staticmethod
    def wait_save_button_press(timeout=600):
        if not USE_BUTTON:
            # raw_input('Enter to continue')
            return 1, "Press save button step skipped"
        try:

            rospy.wait_for_message("/niryo_robot/blockly/save_current_point", Int32, timeout=timeout)
            return 1, "Press detected"
        except rospy.ROSException:

            return -1, "Timeout: no press detected"

    def move_and_compare(self, target, precision_decimal=1):
        status, message = self.__robot.move_joints(*target)
        if status >= 0:
            current_joints = self.__robot.get_joints()
            if not almost_equal_array(self.__robot.get_joints(), target, decimal=precision_decimal):
                raise TestFailure("Target not reached - Actual {} - Target {}".format(current_joints, target))
        return status, message

    def move_and_compare_without_moveit(self, target, precision_decimal=1, duration=4):
        _status, _message = self.__robot.move_without_moveit(target, duration)
        start_time = rospy.Time.now()

if __name__ == '__main__':
    rospy.init_node('niryo_test_FQC_ros_wrapper')
    robot = NiryoRosWrapper()

    LOOPS = 2
    CALIBRATION_LOOPS = 1
    SPIRAL_LOOPS = 2

    USE_VOCAL = True
    USE_BUTTON = True
    SPEED = 100  # %
    ACCELERATION = 100  # %
    FULL = 3

    robot.sound.play('ready.wav')

    print("----- START -----")
    test = TestProduction()
    prog = test.run()
    print("----- END -----")

    test.print_report()

    try:
        set_setting = rospy.ServiceProxy('/niryo_robot_database/settings/set', SetSettings)
```

```
        set_setting('test_report_done', 'True', 'bool')
        try:
            test.send_report()
        except Exception as _e:
            rospy.logerr(f'Failed to send report: {_e}')
        rospy.sleep(3)
    except Exception as _e:
        rospy.logerr(_e)
    if not prog:
        raise TestFailure('Program failure : {}'.format(prog))
```

Folder robot

35 printable files

(file list disabled)

robot__init__.py

```

```

robot\calibration.py

```
# !/usr/bin/env python3

import rospy
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper
from niryo_robot_status.msg import RobotStatus

def _send_fake_running_status():
  """
  Emulate a new status to get access of led ring.

  Because when the 'learning trajectory' mode is finished, the led ring controller is not released, even after calibration.
  """
  robot_status_msg = RobotStatus()
  robot_status_msg.robot_status = RobotStatus.RUNNING_AUTONOMOUS
  robot_status_msg.robot_status_str = 'Running autonomous'
  robot_status_msg.robot_message = ''
  robot_status_msg.logs_status = RobotStatus.NONE
  robot_status_msg.logs_status_str = 'None'
  robot_status_msg.logs_message = ''
  robot_status_msg.out_of_bounds = False
  robot_status_msg.rpi_overheating = False
  rospy.Publisher('/niryo_robot_status/robot_status', RobotStatus, latch=True, queue_size=10).publish(robot_status_msg)

def new_calibration(robot: NiryoRosWrapper):
  robot.clear_collision_detected()
  robot.request_new_calibration()
  robot.calibrate_auto()
  rospy.sleep(0.1)
  _send_fake_running_status()
  rospy.sleep(0.1)
  robot.led_ring.alternate([[248, 252, 0], [0, 0, 248]], 0.5, 5)
  robot.update_tool()
  rospy.sleep(0.1)
  robot.grasp_with_tool()
  rospy.sleep(0.1)
  robot.release_with_tool()
  robot.led_ring.flashing([0, 0, 248], 0.5, 3)
  robot.sound.play("connected.wav")

if __name__ == '__main__':
  rospy.init_node('niryo_test_FQC_ros_wrapper')
  robot = NiryoRosWrapper()
  new_calibration(robot)

```

robot\main.py

```
# First, parse CLI
from util.logger import Log, Logger, LogLevel
from util.cli import CliParser

cli_args = CliParser().parse_args()

# And change settings of logging
Log.write_logs = not cli_args.nowritelogs
Log.folder_name = cli_args.name
```

```
Log.logs_path = cli_args.logspath
Log.use_colors = not cli_args.nocolors
if cli_args.loglevel != Log.level.name:
  for l in LogLevel:
    if l.name == cli_args.loglevel:
      Log.info("Log level set to '@'", l.name)
      Log.level = l
      break
  else: Log.warn("Loglevel '@' not found, used default level", cli_args.loglevel)

Log.info("Importing libraries")
import rospy, signal
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

from util.rospy_logger import redirect_rospy_logger
from util.shell import ShellManager
from util.lock import LockFile
from util.vars import (
  EXIST_REQUESTED_EVENT,
  JOINTS_DEFAULT_POSE,
  TRAJECTORY_DIR_PATH,
  ROSPY_NODE_WAIT_TIMEOUT,
  LOCK_FILE_PATH
)
from calibration import new_calibration
from trajectory.manager import TRAJECTORY_MANAGER
from modbus.server import ModbusServer


class NiryoRobot:
  def __init__(self):
    self.logger = Logger(__class__.__name__)
    self._already_stopped = False
    self.error_occur = False


    self.logger.info("")
    self.logger.info("&fb&lb┌──────────────────────────────────┐")
    self.logger.info("&fb&lb│&fr      Robot is starting       &fb&lb│")
    self.logger.info("&fb&lb│&fr  Please wait one minute ...  &fb&lb│")
    self.logger.info("&fb&lb└──────────────────────────────────┘")
    self.logger.info("")

    self.logger.info("Starting RosPy")
    rospy.init_node("niryo_robot_project")

    self.acquire_lockfile()
    if self.stopped_prematurely(): return

    self.logger.info("Connecting to robot")
    self.robot = NiryoRosWrapper()
    self.terminal: ShellManager = None
    self.modbus: ModbusServer = None

    # callback when another node is registered
    def shutdown_hook(reason=""):
      if EXIST_REQUESTED_EVENT.is_set(): return
      self.logger.info(reason.capitalize() if "shutdown request" in reason else ("Recieved shutdown request"
+ (" with reason: @" if reason else '')), reason)
      reason = reason.lower()

      if "new node registered with same name" in reason:
        self.logger.warn("!! Detected new instance of program !!")
        self.logger.info("Shutting down this program ...")

      elif "program exit" in reason: self.logger.warn("Program exit requested!")
      elif "signal-2" in reason: self.logger.warn("KeyboardInterrupt (CTRL+C) requested!")

      lockfile.notify_release()
      if self.status_sounds:
        try: self.robot.sound.play("warn.wav")
```

```
80          except BaseException as e: self.logger.err(e, no_stacktrace=True)
81        self.stop()
82        return
83
84        self.logger.warn("Ignored shutdown request!")
85      rospy.core.add_preshutdown_hook(shutdown_hook)
86
87    def acquire_lockfile(self):
88      # lock file exist, so probably another handler is present
89      if lockfile.present:
90        self.logger.info("Waiting for the other handler, to stop itself ...")
91        # wait a little
92        EXIST_REQUESTED_EVENT.wait(3)
93
94        # if the lock file is still present and the value is not 0,
95        # we consider that there are no other handlers
96        if lockfile.present:
97          if lockfile.handled():
98            self.logger.warn("Timeout, waiting for other handle (considering there is no other handler)")
99            lockfile.acquire(self)
100           return
101
102         # wait for lock file to be removed by other handler
103         count = ROSPY_NODE_WAIT_TIMEOUT
104         while count:
105           EXIST_REQUESTED_EVENT.wait(1)
106           count -= 1
107           if not lockfile.present:
108             lockfile.acquire(self)
109             return
110
111         if lockfile.present:
112           self.logger.warn("Timeout, waiting for other handle (considering there is no other handler)")
113           lockfile.acquire(self)
114           return
115
116     # probably no other handler, so continue
117     else: lockfile.acquire(self)
118
119   def stopped_prematurely(self):
120     if self.error_occur or EXIST_REQUESTED_EVENT.is_set() or rospy.is_shutdown():
121       self.error_occur = True
122       self.logger.err("\n!! Exited before finish starting")
123       EXIST_REQUESTED_EVENT.set() # make sure the event is triggered
124       self.stop()
125       return True
126     return False
127
128   # TODO: make sound error
129
130   def stop(self):
131     if self._already_stopped: return
132     self._already_stopped = True
133     self.logger.info("Stopping robot ...")
134     EXIST_REQUESTED_EVENT.set()
135     if self.terminal:
136       self.terminal.join(5)
137       self.terminal = None
138     if self.modbus:
139       self.modbus.stop()
140       self.modbus = None
141     TRAJECTORY_MANAGER.save()
142
143     # Silently disable shutdown flag
144     if self.status_sounds:
145       shutdown = rospy.core._shutdown_flag
146       rospy.core._shutdown_flag = False
147       try:
148         if self.error_occur:
149           self.robot.sound.play("warn.wav")
```

```
150           self.robot.sound.play("error.wav")
151         else: self.robot.sound.play("disconnected.wav")
152       except BaseException as e: self.logger.err(e, no_stacktrace=True)
153       rospy.core._shutdown_flag = shutdown
154 
155     lockfile.release()
156     self.logger.info("Robot stopped")
157 
158   def stop_complete(self):
159     return (self._already_stopped and EXIST_REQUESTED_EVENT.is_set() and
160         #    rospy.is_shutdown() and
161             self.terminal is None and self.modbus is None)
162 
163   def start(self, do_calibration=True, load_trajectories=True, start_modbus=True, use_terminal=True,
    status_sounds=True, default_pose=True):
164     if self.stopped_prematurely(): return
165     self.status_sounds = status_sounds
166 
167     if do_calibration:
168       self.logger.info("Started calibration")
169       try: new_calibration(self.robot)
170       except Exception as e:
171         self.logger.err(e, no_stacktrace=True)
172         self.error_occur = True
173       else: self.logger.info("Calibration finished")
174 
175       if self.stopped_prematurely(): return
176 
177     if load_trajectories:
178       self.logger.info("Loading trajectories ...")
179       try: TRAJECTORY_MANAGER.load(TRAJECTORY_DIR_PATH).sort()
180       except Exception as e:
181         self.logger.err(e)
182         self.error_occur = True
183 
184       if self.stopped_prematurely(): return
185 
186     if start_modbus:
187       self.logger.info("Starting Modbus/TCP server ...")
188       try: self.modbus = ModbusServer(self.robot)
189       except:
190         self.error_occur = True
191         self.stop()
192       else: self.modbus.start()
193 
194       if self.stopped_prematurely(): return
195 
196     if use_terminal:
197       self.logger.info("Starting terminal ...")
198       self.terminal = ShellManager(self.robot)
199 
200     if EXIST_REQUESTED_EVENT.wait(1) and self.stopped_prematurely(): return
201     self.logger.info("")
202     self.logger.info("&fb&lb┌──────────────────────────────┐")
203     self.logger.info("&fb&lb│&fr        Robot started!        &fb&lb│")
204     self.logger.info("&fb&lb│&fr      Now enjoy the fun.      &fb&lb│")
205     self.logger.info("&fb&lb└──────────────────────────────┘")
206     self.logger.info("")
207 
208     if use_terminal: self.terminal.start()
209 
210     try:
211       if self.status_sounds: self.robot.sound.play("ready.wav")
212       self.setup()
213       while not EXIST_REQUESTED_EVENT.is_set():
214         self.loop()
215         EXIST_REQUESTED_EVENT.wait(0.01) # No need to run at full speed
216 
217       # Go to default position
218       if default_pose and not rospy.is_shutdown() and not self._already_stopped:
219         if self.robot.collision_detected:
```

```
          self.logger.warn("Collision detected during last movement.")
          self.logger.warn("Robot will not move to the default point, to avoid another potential collision."
)
        else:
          self.logger.info("Moving to default position ...")
          self.robot.move_joints(*JOINTS_DEFAULT_POSE)

    except Exception as e:
      if type(e) not in (KeyboardInterrupt, SystemExit):
        self.logger.err(e)
        self.error_occur = True
    finally: self.stop()

  def setup(self):
    """Setup function of program, will be called one time. Put the variable creation or other things here.""
"
    ...

  def loop(self):
    """Main loop of program, will be called every iteration. Must not be a blocking method."""
    ...


if __name__ == '__main__':
  lockfile = LockFile(LOCK_FILE_PATH)
  signal.signal(signal.SIGINT|signal.SIGTERM|signal.SIGKILL|signal.SIGQUIT, lockfile.release)

  try:
    redirect_rospy_logger()

    main = NiryoRobot()
    main.start(not cli_args.nocalibration, not cli_args.notrajectories, not cli_args.nomodbus,
               not cli_args.noterminal, not cli_args.nostatussound, not cli_args.nodefaultpose)
    count = ROSPY_NODE_WAIT_TIMEOUT * 10
    while count and not main.stop_complete():
      rospy.sleep(0.1)
      count -= 1
    if not main.stop_complete(): main.logger.warn("Timeout stopping robot, force exiting!")
    main.logger.info("Stopping RosPy ...")
    rospy.signal_shutdown("program exit")

  except BaseException as e:
    if type(e) not in (KeyboardInterrupt, SystemExit):
      from util.logger import Log

      Log.err("\n!! FATAL ERROR !!\n")
      Log.err(e)


  finally:
    lockfile.release()

```

robot\modbus__init__.py

```

```

robot\modbus\holding_register.py

```
"""
Re-implementation of niryo_robot_modbus.holding_register_data_block, for own use
"""

import rospy, threading

from niryo_robot_modbus.data_block import DataBlock
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper
from niryo_robot_msgs.msg import CommandStatus

from util.logger import Logger
from trajectory.learning import TrajectoryLearning
from trajectory.player import TrajectoryPlayer
```

```python
from trajectory.manager import TRAJECTORY_MANAGER
from trajectory.parser import TrajectoryParser
from trajectory.step.goal import GoalFailure


"""
 - Each address contains a 16 bits value
 - READ/WRITE registers

 --> Used to give commands to the robot
 ( ! the stored values correspond to the last given command,
 not the current robot state !)
"""
HR_LEARNING_MODE = 0              # 0: disable, 1: enable
HR_MAKE_FLAG = 1                  # 0: nothing, 1: add position flag
HR_TOOL_STATE = 2                 # 0: disable, 1: enable
HR_MOTOR_SPEED = 3                # 1-200

HR_SELECTED_TRAJECTORY_NAME = 10
HR_SELECTED_TRAJECTORY_ID = 31
TRAJECTORY_NAME_SIZE = 20

HR_SAVE_DEL_TRAJECTORY = 40       # 0: nothing, 1: save selected, 2: delete selected
HR_SELECT_TRAJECTORY_INDEX = 41   # 0: unselect
HR_RUN_SELECTED_TRAJECTORY = 42   # 0: waiting, 1: running
HR_STOP_RUNNING_TRAJECTORY = 43   # 0: nothing, 1: stop

HR_COUNT_SAVED_TRAJECTORY = 50    # 0: no saved trajectories
HR_HAS_TRAJECTORY_SELECTED = 51   # 0: no, 1: yes

HR_ERROR_STATUS = 100             # 0: no error

# Error codes
ERROR_UNKNOWN = 100  # logs need to be readed for that
ERROR_NOT_LEARNING = 101
ERROR_NOTHING_SELECTED = 102
ERROR_NOT_FOUND = 103
ERROR_INVALID_NAME = 104
ERROR_EMPTY_NAME = 105
ERROR_INVALID_TRAJECTORY = 106
ERROR_NOT_RUNNING = 107
ERROR_ALREADY_RUNNING = 108
ERROR_TRAJECTORY_ABORDED = 109


class HoldingRegisterDataBlock(DataBlock):
  def __init__(self, robot: NiryoRosWrapper):
    super().__init__()
    self.robot = robot
    self.logger = Logger(__class__.__name__)
    self.learning = TrajectoryLearning(robot)
    self.playing = None
    self.selected = 0

    self.execution_thread = threading.Thread()
    self.is_action_client_running = False

    # update value for first time
    # TODO: find a way to mak this synchonized
    self.setValuesOffset(HR_COUNT_SAVED_TRAJECTORY, TRAJECTORY_MANAGER.count_trajectories())

  def __set_move_in_progress(self):
    self.setValuesOffset(HR_STOP_RUNNING_TRAJECTORY, 0)
    self.setValuesOffset(HR_RUN_SELECTED_TRAJECTORY, 1)

  def __set_error_code(self, code):
    self.setValuesOffset(HR_ERROR_STATUS, code)

  def __set_move_done(self, status_result):
    self.__set_error_code(self.__get_status_code(status_result))
```

```
    self.setValuesOffset(HR_STOP_RUNNING_TRAJECTORY, 0)
    self.setValuesOffset(HR_RUN_SELECTED_TRAJECTORY, 0)

  def __run_action_async(self, func, *args):
    if self.check_move_running(): return

    def wrapper():
      self.is_action_client_running = True
      self.__set_move_in_progress()

      try:
        try: response = func(*args)
        except RuntimeError: self.__set_move_done(0)
        except Exception as e:
          if type(e) == RuntimeError and "generator raised StopIteration" in str(e): self.__set_move_done(ERROR_TRAJECTORY_ABORDED)
          else:
            self.logger.err("ModbusAsyncAction - "+type(e).__name__+": "+str(e))
            error = str(e).strip()
            if error and "error code:" in error.lower():
              code = error.split('\n')[0]
              code = code[code.find(':')+1].strip()
              self.update_status(int(code) if code else None)
            elif "clear_collision_detected" in error: self.__set_move_done(9)
            else: self.update_status(None)
        else: self.update_status(response)
      except BaseException as e:
        self.__set_error_code(ERROR_UNKNOWN)
        self.logger.err(e, no_stacktrace=True)
      self.is_action_client_running = False

    self.execution_thread = threading.Thread(name="ModbusAsyncAction", target=wrapper, daemon=True)
    self.execution_thread.start()

  def __get_status_code(self, status):
      if type(status) in (tuple, list): return status[0]
      elif type(status) == str and status: return int(status)
      else:
        s = getattr(status, "status", None)
        if s is not None: return s
      return status

  def setValuesOffset(self, address, values):
    super().setValues(address+1, values if type(values) == list else [values])


  def check_move_running(self):
    if self.is_action_client_running or self.execution_thread.is_alive():
      self.__set_error_code(ERROR_ALREADY_RUNNING)
      return True
    return False

  def update_status(self, status):
    self.setValuesOffset(HR_STOP_RUNNING_TRAJECTORY, 0)
    self.setValuesOffset(HR_RUN_SELECTED_TRAJECTORY, 0)
    status = self.__get_status_code(status)

    if status == CommandStatus.NO_CONNECTION or status == None:
      self.__set_error_code(7)
    elif status == CommandStatus.GOAL_TIMEOUT:
      self.__set_error_code(6)
    elif status == CommandStatus.REJECTED:
      self.__set_error_code(2)
    elif status == CommandStatus.ABORTED:
      self.__set_error_code(3)
    elif status == CommandStatus.PREEMPTED:
      self.__set_error_code(4)
    elif status != CommandStatus.SUCCESS:
      self.__set_error_code(5)
    else:
      self.__set_error_code(1)
```

```
154
155   def get_trajectory_name(self):
156     name = ''.join([chr(i) for i in self.getValuesOffset(HR_SELECTED_TRAJECTORY_NAME, TRAJECTORY_NAME_SIZE)
    if i])
157     if not name:
158       self.__set_error_code(ERROR_EMPTY_NAME)
159       return None
160     elif not name.isprintable():
161       self.__set_error_code(ERROR_INVALID_NAME)
162       return None
163     return name
164
165   def is_trajectory_selected(self):
166     return self.getValuesOffset(HR_HAS_TRAJECTORY_SELECTED)[0] != 0
167
168   def setValues(self, address, values):
169     try:
170       self.process_command(address, values)
171   #    super().setValues(address, values)
172     except BaseException as e:
173       self.logger.err(e, no_stacktrace=True)
174       self.update_status(None)
175
176   def process_command(self, address, values):
177     address -= 1
178     if len(values) == 0:
179         return
180
181     elif address == HR_LEARNING_MODE:
182       self.set_learning_mode(values[0])
183     elif address == HR_MAKE_FLAG:
184       self.add_pose_flag(values[0])
185     elif address == HR_TOOL_STATE:
186       self.set_tool_state(values[0])
187     elif address == HR_MOTOR_SPEED:
188       self.set_motor_speed(values[0])
189
190     elif address == HR_SELECTED_TRAJECTORY_ID:
191       ... # id is unique, so changes are not allowed
192     elif HR_SELECTED_TRAJECTORY_NAME <= address <= HR_SELECTED_TRAJECTORY_NAME+TRAJECTORY_NAME_SIZE:
193       # ensure name is 20 letters max
194       self.setValuesOffset(address, values[0:TRAJECTORY_NAME_SIZE-(address-HR_SELECTED_TRAJECTORY_NAME)])
195
196     elif address == HR_SAVE_DEL_TRAJECTORY:
197       self.save_delete_trajectory(values[0])
198     elif address == HR_SELECT_TRAJECTORY_INDEX:
199       self.select_trajectory(values[0])
200     elif address == HR_RUN_SELECTED_TRAJECTORY:
201       self.run_selected_trajectory(values[0])
202     elif address == HR_STOP_RUNNING_TRAJECTORY:
203       self.stop_running_trajectory(values[0])
204
205     elif address == HR_COUNT_SAVED_TRAJECTORY:
206       ... # read only number
207     elif address == HR_HAS_TRAJECTORY_SELECTED:
208       self.unselect_trajectory(values[0])
209
210     elif address == HR_ERROR_STATUS:
211       ... # this is an error status, no changes are allowed
212
213
214   def set_learning_mode(self, value):
215     self.setValuesOffset(HR_LEARNING_MODE, int(value != 0))
216
217     if value == 0:
218       if self.learning.started:
219         self.learning.stop()
220
221     else:
222       self.learning.clear()
223       self.learning.start()
```

```
      self.tool_state = self.learning.trajectory.get_start_state()[0]
      self.setValuesOffset(HR_TOOL_STATE, int(self.tool_state))
      index = TRAJECTORY_MANAGER.count_trajectories()+1
      self.setValuesOffset(HR_SELECT_TRAJECTORY_INDEX, index)
      self.setValuesOffset(HR_SELECTED_TRAJECTORY_ID, self.learning.trajectory.get_id())
      self.setValuesOffset(HR_SELECTED_TRAJECTORY_NAME, [0]*20)
      self.setValuesOffset(HR_HAS_TRAJECTORY_SELECTED, 1)
      self.selected = index

    self.__set_error_code(0)

  def add_pose_flag(self, value):
    self.setValuesOffset(HR_MAKE_FLAG, int(value != 0))

    if value:
      if not self.learning.started: self.__set_error_code(ERROR_NOT_LEARNING)
      else: self.learning.add_pos_flag()

    self.setValuesOffset(HR_MAKE_FLAG, 0)
    self.__set_error_code(0)

  def set_tool_state(self, value):
    self.setValuesOffset(HR_TOOL_STATE, int(value != 0))

    if not self.learning.started:
      self.__set_error_code(ERROR_NOT_LEARNING)
      return
    try: self.learning.set_tool_state(value != 0)
    except GoalFailure as e:
      self.logger.err(e)
      self.__set_error_code(e.status)
    else: self.__set_error_code(0)

  def set_motor_speed(self, value):
    self.setValuesOffset(HR_MOTOR_SPEED, int(value))

    if not self.learning.started:
      self.__set_error_code(ERROR_NOT_LEARNING)
      return
    try: self.learning.set_motor_speed(value)
    except GoalFailure as e:
      self.logger.err(e)
      self.__set_error_code(e.status)
    else: self.__set_error_code(0)

  def save_delete_trajectory(self, value):
    self.setValuesOffset(HR_SAVE_DEL_TRAJECTORY, value)

    if value and not self.is_trajectory_selected():
      self.__set_error_code(ERROR_NOTHING_SELECTED)

    elif value == 1:
      if self.learning.trajectory and not self.learning.learn_saved:
        name = self.get_trajectory_name()
        if name:
          name = + str(self.learning.trajectory.get_id()) + '-' + name
          TRAJECTORY_MANAGER.add(self.learning.save(name))
          self.setValuesOffset(HR_COUNT_SAVED_TRAJECTORY, TRAJECTORY_MANAGER.count_trajectories())
          self.unselect_trajectory(0)
          self.__set_error_code(0)

      elif self.is_trajectory_selected():
        name = self.get_trajectory_name()
        if name:
          try:
            traj = TRAJECTORY_MANAGER.get(self.selected)
            traj.set_name(name)
            TrajectoryParser.write(traj)
            self.unselect_trajectory(0)
            self.__set_error_code(0)
```

```
294           except IndexError: self.__set_error_code(ERROR_NOT_FOUND)
295           except BaseException as e:
296             self.__set_error_code(ERROR_UNKNOWN)
297             self.logger.err(e, no_stacktrace=True)
298 
299       else: self.__set_error_code(ERROR_NOTHING_SELECTED)
300 
301     elif value == 2:
302       if self.learning.trajectory and not self.learning.learn_saved:
303         self.unselect_trajectory(0)
304 
305       elif self.is_trajectory_selected():
306         try:
307           traj = TRAJECTORY_MANAGER.get(self.selected)
308           TRAJECTORY_MANAGER.delete(traj)
309           self.setValuesOffset(HR_COUNT_SAVED_TRAJECTORY, TRAJECTORY_MANAGER.count_trajectories())
310           self.unselect_trajectory(0)
311           self.__set_error_code(0)
312         except IndexError: self.__set_error_code(ERROR_NOT_FOUND)
313         except BaseException as e:
314           self.__set_error_code(ERROR_UNKNOWN)
315           self.logger.err(e, no_stacktrace=True)
316 
317       else: self.__set_error_code(ERROR_NOTHING_SELECTED)
318 
319     self.setValuesOffset(HR_SAVE_DEL_TRAJECTORY, 0)
320 
321   def select_trajectory(self, value):
322     self.setValuesOffset(HR_SELECT_TRAJECTORY_INDEX, value)
323 
324     try:
325       traj = TRAJECTORY_MANAGER.get(value)
326       error = False
327 
328       start = traj.get_start_state()
329       if error or start is None: error = True
330       else: self.setValuesOffset(HR_TOOL_STATE, int(start[0]))
331 
332       id = traj.get_id()
333       if error or id is None: error = True
334       else: self.setValuesOffset(HR_SELECTED_TRAJECTORY_ID, id)
335 
336       if not error:
337         name = traj.get_name()
338         self.setValuesOffset(HR_SELECTED_TRAJECTORY_NAME, [ord(name[i]) if i < len(name) else 0 for i in
    range(TRAJECTORY_NAME_SIZE)])
339 
340       if error:
341         self.unselect_trajectory(0)
342         self.__set_error_code(ERROR_INVALID_TRAJECTORY)
343       else:
344         self.selected = value
345         self.__set_error_code(0)
346         self.setValuesOffset(HR_HAS_TRAJECTORY_SELECTED, 1)
347     except: self.__set_error_code(ERROR_NOT_FOUND)
348 
349   def run_selected_trajectory(self, value):
350     self.setValuesOffset(HR_RUN_SELECTED_TRAJECTORY, int(value != 0))
351 
352     if value:
353       if self.is_trajectory_selected():
354         self.__set_error_code(ERROR_NOTHING_SELECTED)
355         return
356 
357       try:
358         def wrapper():
359           self.playing = TrajectoryPlayer(self.robot, TRAJECTORY_MANAGER.get(self.selected))
360           response = self.playing.play()
361           self.playing = None
362           return response
363         self.__run_action_async(wrapper)
```

```
        self.__set_error_code(0)
      except: self.__set_error_code(ERROR_NOT_FOUND)

  def stop_running_trajectory(self, value):
    self.setValuesOffset(HR_STOP_RUNNING_TRAJECTORY, int(value != 0))

    if value:
      if self.playing is None or not self.is_action_client_running or not self.execution_thread.is_alive():
        self.__set_error_code(ERROR_NOT_RUNNING)
        return

      if self.playing:
        try:
          self.playing.stop()
          self.playing = None
          self.__set_error_code(0)
        except: self.__set_error_code(ERROR_NOT_RUNNING)

    self.setValuesOffset(HR_STOP_RUNNING_TRAJECTORY, 0)

  def unselect_trajectory(self, value):
    if value == 0:
      self.setValuesOffset(HR_SELECTED_TRAJECTORY_NAME, [0]*TRAJECTORY_NAME_SIZE)
      self.setValuesOffset(HR_SELECTED_TRAJECTORY_ID, 0)
      self.setValuesOffset(HR_SELECT_TRAJECTORY_INDEX, 0)
      self.setValuesOffset(HR_HAS_TRAJECTORY_SELECTED, 0)
      self.selected = 0

```

robot\modbus\input_register.py

```
"""
Re-implementation of niryo_robot_modbus.input_register_data_block, for own use
"""

from niryo_robot_modbus.data_block import DataBlock
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

from util.logger import Logger

"""
 - Each address contains a 16 bits value
 - READ ONLY registers

 --> State of the robot
"""
...


def handle_negative(val):
    """
    Positive number : 0 - 32767
    Negative number : 32768 - 65535
    """
    if val < 0:
        val = (1 << 15) - val
    return val


class InputRegisterDataBlock(DataBlock):
  def __init__(self, robot: NiryoRosWrapper):
    super().__init__()
    self.robot = robot
    self.logger = Logger(__class__.__name__)

  ...

  # The input register is not needed, so i will keep the class empty

```

robot\modbus\server.py

```python
"""
Re-implementation of niryo_robot_modbus.modbus_server, for own use
"""

from util.logger import Logger
from util.errors import ModbusServerError
from util.vars import (
  MODBUS_SERVER_ADDRESS,
  MODBUS_SERVER_PORT,
  MODBUS_IDENTITY_VENDOR_NAME,
  MODBUS_IDENTITY_VENDOR_URL,
  MODBUS_IDENTITY_PRODUCT_NAME,
  MODBUS_IDENTITY_MODEL_NAME,
  MODBUS_IDENTITY_REVISION
)

from threading import Thread

from pymodbus.server.sync import ModbusTcpServer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext

from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

from .input_register import InputRegisterDataBlock
from .holding_register import HoldingRegisterDataBlock

__all__ = [
  "ModbusServer"
]


class ModbusServer:
  def __init__(self, robot: NiryoRosWrapper):
    self.logger = Logger(__class__.__name__)
    self.thread = None

    self.input_register = InputRegisterDataBlock(robot)
    self.holding_register = HoldingRegisterDataBlock(robot)
    self.store = ModbusSlaveContext(hr=self.holding_register, ir=self.input_register)
    self.context = ModbusServerContext(slaves=self.store, single=True)

    self.identity = ModbusDeviceIdentification()
    self.identity.VendorName = MODBUS_IDENTITY_VENDOR_NAME
    self.identity.VendorUrl = MODBUS_IDENTITY_VENDOR_URL
    self.identity.ProductName = MODBUS_IDENTITY_PRODUCT_NAME
    self.identity.ModelName = MODBUS_IDENTITY_MODEL_NAME
    self.identity.MajorMinorRevision = MODBUS_IDENTITY_REVISION

    try: self.server = ModbusTcpServer(context=self.context, framer=None, identity=self.identity,
                                       address=(MODBUS_SERVER_ADDRESS, MODBUS_SERVER_PORT),
                                       allow_reuse_address=True)
    except OSError as err:
      message = "TCP server unable to start: " + str(err)
      self.logger.err(message)
      raise ModbusServerError(message)
    self.logger.info("Created server")

  def start(self):
    if self.thread and self.thread.is_alive(): raise ModbusServerError("server already started")
    self.thread = Thread(target=self.server.serve_forever, name="NiryoModbusServer", daemon=True)
    self.thread.start()
    self.logger.info("Started server")

  def stop(self):
    if not self.thread  or not self.thread.is_alive(): raise ModbusServerError("server not started")
    self.logger.info("Closing server")
    self.server.shutdown()
    self.thread.join(10)
    self.server.server_close()
```

```
    if not self.thread.is_alive(): self.logger.info("Stopped server")
    else: self.logger.warn("Timeout when stopping server")
    self.thread = None

```

robot\trajectory__init__.py

```

```

robot\trajectory\controller.py

```
from .step.type import *
from .step.value import *
from .step.goal import GoalReport
from util.vars import TRAJECTORY_NEEDED_STEPS
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

__all__ = [
  "TrajectoryStep",
  "TrajectoryController",
]


class TrajectoryStep:
  def __init__(self, type: StepType, value: StepValue=StepValue()):
    self.type = type
    self.value = value

  def __repr__(self):
    return f"{__class__.__name__}(step={repr(self.type)}, value={repr(self.value)})"

  __str__ = __repr__

  def serialize(self) -> str:
    return self.type.serialize(self.value)

  def deserialize(self, buffer: str):
    self.value = self.type.deserialize(buffer)

  def run(self, robot: NiryoRosWrapper) -> 'GoalReport':
    return self.type.executor.run(robot, self.value)

  def get_value(self) -> object:
    return self.value.value if self.value else self.value

  def is_property(self) -> bool:
    return self.type.is_property


class PropertyStep(TrajectoryStep):
  """This is same as TrajectoryStep"""
  ...


class TrajectoryController:
  def __init__(self):
    self.__properties = {}
    self.__steps = []

  def __iter__(self):
    return iter(self.get_content())

  def set_property(self, step: PropertyStep) -> 'TrajectoryController':
    if not step.is_property(): raise ValueError("step type is not an property. Use .add_step() to add normal steps")
    self.__properties.update({step.type: step})
    return self

  def get_property(self, type: StepType) -> PropertyStep:
    return self.__properties.get(type, None)

  def add_step(self, step: TrajectoryStep) -> 'TrajectoryController':
```

```
    if step.is_property(): raise ValueError("this step is an property. Use .set_property() to set info steps"
)
    if not step.type.duplicate_allowed and self.contains(step.type): raise ValueError(f"step
{step.type.alias!r} already exist, and does not allow more than one use.")
    self.__steps.append(step)
    return self

  def get_step(self, index: int) -> TrajectoryStep:
    return self.__steps[index]

  def pop_step(self, index: int=-1) -> TrajectoryStep:
    return self.__steps.pop(index)

  def clear(self) -> 'TrajectoryController':
    self.__init__(self)
    return self

  def contains(self, type: StepType) -> bool:
    for s in self.__steps:
      if s.type == type: return True
    return False

  def count_steps(self) -> int:
    return len(self.__steps)

  def get_steps(self) -> 'list[PropertyStep]':
    return self.__steps.copy()

  def get_properties(self) -> 'list[TrajectoryStep]':
    return list(self.__properties.values())

  def get_content(self) -> 'list[TrajectoryStep]':
    for s in TRAJECTORY_NEEDED_STEPS:
      if s not in self.__properties and not self.contains(s):
        raise ValueError(f"{s.label} step ({s.alias!r}) is needed, but not present")
    if self.count_steps() == 0: raise IndexError("no usefull steps in trajectory")
    return self.get_properties() + self.get_steps()

```

robot\trajectory\executor.py

```
from .parser import TrajectoryParser
from .player import TrajectoryPlayer

from util.errors import TrajectoryPlayError
from util.vars import get_trajectory_file

from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

__all__ = [
  "TrajectoryExecutor"
]


class TrajectoryExecutor(TrajectoryPlayer):
  def __init__(self, robot: NiryoRosWrapper, name: str, ignore_errors: bool=False):
    super().__init__(robot, TrajectoryParser.read(get_trajectory_file(name)), ignore_errors)

```

robot\trajectory\learning.py

```
from util.logger import Logger
from util.errors import TrajectoryLearningError
from util.strings import get_child_path

from .trajectory import Trajectory
from .parser import TrajectoryParser
from .step.goal import GoalReport

from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

__all__ = [
```

```
  "TrajectoryLearningError",
  "TrajectoryLearning"
]


class TrajectoryLearning:
  def __init__(self, robot: NiryoRosWrapper):
    self.__logger = Logger(__class__.__name__)
    self.robot = robot
    self.__started = False
    self.__saved = False
    self.trajectory: Trajectory = None
    self.__deep = False

  def ensure_started(self):
    if not self.__started: raise TrajectoryLearningError("learning mode not started")

  def ensure_stopped(self):
    if self.__started: raise TrajectoryLearningError("stop learning mode first")

  def start(self, deep_learning=False):
    if self.__started: raise TrajectoryLearningError("learning mode already started")
    self.__deep = deep_learning
    self.__started = True
    if self.__deep: self.robot.set_learning_mode(True)
    if self.trajectory: self.__logger.warn("previous trajectory not saved or cleared, overriding.")
    self.__saved = False
    self.robot.release_with_tool()
    self.trajectory = Trajectory().set_start_state(False, self.get_robot_pos()).new_id()

  def stop(self, no_check=False) -> Trajectory:
    if not no_check: self.ensure_started()
    if self.__deep: self.robot.set_learning_mode(False)
    self.__deep = False
    self.__started = False
    self.__saved = False
    return self.trajectory

  def save(self, trajectory_name: str, custom_filename: str=None, no_check=False) -> Trajectory:
    if not trajectory_name: raise ValueError("trajectory name is empty")
    if not no_check: self.ensure_stopped()
    if not self.trajectory: raise TrajectoryLearningError("learning was never started")
    name = custom_filename or str(self.trajectory.get_id()) + '-' + trajectory_name
    self.trajectory.set_name(trajectory_name)
.set_filename(get_child_path(TrajectoryParser.make_trajectory_file_by_name(name)))
    TrajectoryParser.write(self.trajectory)
    traj = self.trajectory
    self.trajectory = None
    self.__saved = True
    return traj

  def clear(self):
    deep = self.__deep
    started = self.__started
    self.stop(no_check=True)
    self.trajectory = None
    if started: self.start(deep)

  def get_robot_pos(self) -> 'tuple[float]':
    return self.robot.get_joints()

  def add_pos_flag(self):
    self.ensure_started()
    self.trajectory.add_joints(self.get_robot_pos())

  def set_tool_state(self, enabled: bool):
    self.ensure_started()
    GoalReport(self.robot.grasp_with_tool if enabled else self.robot.release_with_tool).raise_if_error()
    self.trajectory.add_tool_state(enabled)

  def set_motor_speed(self, percent: int):
```

```
    self.ensure_started()
    GoalReport(self.robot.set_arm_max_velocity, percent).raise_if_error()
    self.trajectory.add_motor_speed(percent)

  @property
  def started(self):
    return self.__started

  @property
  def deep_learning(self):
    return self.__deep

  @property
  def learn_saved(self):
    return self.__saved
```

robot\trajectory\manager.py

```
from .trajectory import Trajectory
from .parser import TrajectoryParser

from util.logger import Logger
from util.errors import TrajectoryManagerError
from util.vars import list_trajectory_files

__all__ = [
  "TrajectoryManager",
  "TRAJECTORY_MANAGER",
]


class TrajectoryManager:
  def __init__(self):
    self.trajectories: 'list[Trajectory]' = []
    self.logger = Logger(__class__.__name__)
    self.current_play = None

  def __iter__(self):
    return iter(self.trajectories)

  def count_trajectories(self):
    return len(self.trajectories)

  def load_add(self, custom_directory: str="") -> 'TrajectoryManager':
    try:
      for file in list_trajectory_files(custom_directory):
        self.logger.debug("Loading trajectory: @", file)
        self.add(TrajectoryParser.read(file))
    except Exception as e:
      if isinstance(e, TrajectoryManagerError): raise
      raise TrajectoryManagerError("Error while loading trajectories: " + str(e)) from None
    return self

  def load_replace(self, custom_directory: str="") -> 'TrajectoryManager':
    # clear all ids
    Trajectory.clear_all_ids()
    return self.clear().load_add(custom_directory)

  load = load_add
  reload = load_replace

  def save(self) -> 'TrajectoryManager':
    # Save the error instead of raising it quickly.
    # This way avoid more error when stopping the program.
    error = None
    try:
      for i, t in enumerate(self):
        if not t.get_name() and not t.get_filename():
          error = ValueError(f"no valid properties to save the trajectory {i}. Make sure the name or/and the filename properties are set.")
        TrajectoryParser.write(t)
```

```
        self.logger.debug("Saved trajectory '@' at @", t.get_name(), t.get_filename())
    except Exception as e: error = e

    if error: raise TrajectoryManagerError("Error while saving trajectories (last error): " + str(error)) from None
    return self

  def add(self, trajectory: Trajectory) -> 'TrajectoryManager':
    # Check trajectory id
    id = trajectory.get_id()
    if id is None: raise TrajectoryManagerError("id is missing is trajectory, use trajectory.new_id() to assign an id")
    if self.get_by_id(id) is not None:
      raise TrajectoryManagerError(f"duplicated trajectory with id {id}, use trajectory.new_id() to assign a new id")
    self.trajectories.append(trajectory)
    return self

  def remove(self, trajectory: Trajectory):
    self.trajectories.remove(trajectory)

  def delete(self, trajectory: Trajectory):
    self.remove(trajectory)
    TrajectoryParser.delete(trajectory)
    del trajectory

  def clear(self) -> 'TrajectoryManager':
    self.trajectories.clear()
    return self

  def get(self, index: int) -> Trajectory:
    return self.trajectories[index]

  def get_by_name(self, traj_name: str, ignore_case: bool=True) -> Trajectory:
    if ignore_case:
      traj_name = traj_name.lower()
      return self.find(lambda t: t.get_name().lower() == traj_name)
    return self.find(lambda t: t.get_name() == traj_name)

  def get_by_id(self, id: int) -> Trajectory:
    return self.find(lambda t: t.get_id() == id)

  def find(self, consumer: 'function[[Trajectory], bool]') -> Trajectory:
    for t in self:
      if consumer(t): return t
    return None

  def sort(self) -> 'TrajectoryManager':
    self.trajectories = sorted(self.trajectories, key=lambda v: v.get_id())


# Global Instance
TRAJECTORY_MANAGER = TrajectoryManager()
```

robot\trajectory\parser.py

```
from .trajectory import Trajectory
from .controller import TrajectoryStep, PropertyStep
from .step.type import *

from util.errors import TrajectoryReadError
from util.vars import TRAJECTORY_NEEDED_STEPS
from util.vars import get_trajectory_file, format_trajectory_file_name
from util.strings import sanitize_filename, find_available_file, get_parent_path, get_child_path

import os

__all__ = [
  "TrajectoryParser"
]

```

```
16
17  class TrajectoryParser:
18    @staticmethod
19    def make_trajectory_file_by_name(trajectory_name: str) -> str:
20      if not trajectory_name: raise ValueError("no trajectory name specified")
21      return
find_available_file(get_trajectory_file(format_trajectory_file_name(sanitize_filename(trajectory_name.lower()
.replace(' ', '-')))))
22
23    @staticmethod
24    def write(traj: Trajectory):
25      file = traj.get_filename()
26      if file: file = get_trajectory_file(file)
27      else:
28        # Format a new file name with trajectory name (or default name), if not defined
29        name = traj.get_name()
30        if not name: name = "untitled-"+str(traj.get_id() or "trajectory")
31        file = TrajectoryParser.make_trajectory_file_by_name(name)
32        traj.set_filename(get_child_path(file))
33
34      content = traj.get_content()
35      path = get_parent_path(file)
36      if not os.path.exists(path): os.mkdir(path)
37      with open(file, "wt") as f:
38        for t in content: f.write(t.serialize() + '\n')
39
40    @staticmethod
41    def delete(traj: Trajectory):
42      file = get_trajectory_file(traj.get_filename())
43      if os.path.exists(file): os.remove(file)
44
45    @staticmethod
46    def read(file: str) -> Trajectory:
47      traj = Trajectory()
48      traj.set_filename(get_child_path(file))
49      found_steps = []
50
51      with open(file, "rt") as f:
52        i = 1
53        line = TrajectoryParser.read_next_step(f)
54
55        while line:
56          step = TrajectoryParser.find_step_type(line)
57
58          if not step.duplicate_allowed and step in found_steps:
59            raise TrajectoryReadError(file, f"duplicated {step.label} step ({step.alias!r}) at line {i}. "
60                                      "This step must only appear once in this trajectory.")
61          try:
62            if step.is_property: traj.set_property(PropertyStep(step, step.deserialize(line)))
63            else: traj.add_step(TrajectoryStep(step, step.deserialize(line)))
64          except Exception as e:
65            raise TrajectoryReadError(file, f"unable to decode {step.label} step ({step.alias!r}) at line {i}:
", e) from None
66
67          found_steps.append(step)
68          i += 1
69          line = TrajectoryParser.read_next_step(f)
70
71      if not all(s in found_steps for s in TRAJECTORY_NEEDED_STEPS):
72        steps = ", ".join(f"{s.alias!r} ({s.label})" for s in TRAJECTORY_NEEDED_STEPS if s not in found_steps)
73        raise TrajectoryReadError(file, "missing needed step(s) in trajectory: " + steps)
74      return traj
75
76    @staticmethod
77    def read_next_step(file) -> str:
78      line = file.readline()
79      l = line.strip()
80      while line and not l:
81        line = file.readline()
82        l.strip()
83      return l
```

```

  @staticmethod
  def find_step_type(line: str) -> StepType:
    return StepTypes.find(line)

```

robot\trajectory\player.py

```
from .trajectory import Trajectory
from .manager import TRAJECTORY_MANAGER

from util.errors import TrajectoryPlayError, TrajectoryNotFoundError
from util.logger import Logger

from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper
import typing, rospy, threading

__all__ = [
  "TrajectoryPlayer"
]


class TrajectoryPlayer:
  def __init__(self, robot: NiryoRosWrapper, trajectory: Trajectory, ignore_errors: bool=False):
    self.robot = robot
    self.logger = Logger(__class__.__name__)
    self.trajectory = trajectory
    self.ignore_errors = ignore_errors
    self.play_started = False
    self.thread = None

  @staticmethod
  def play_trajectory(robot: NiryoRosWrapper, id_or_name: 'typing.Union[int, str]', play_async: bool=False) -
> 'TrajectoryPlayer':
    trajectory = TRAJECTORY_MANAGER.get_by_name(id_or_name)
    if not trajectory and id_or_name.isdigit(): trajectory = TRAJECTORY_MANAGER.get_by_id(int(id_or_name))
    if not trajectory: raise TrajectoryNotFoundError(f"Trajectory {id_or_name!r} not found")
    else:
      player = TrajectoryPlayer(robot, trajectory)
      if play_async: player.play_async()
      else: player.play()
      return player

  def is_playing(self) -> bool:
    return self.play_started or (self.thread and self.thread.is_alive())

  def play_all(self):
    if self.is_playing(): raise TrajectoryPlayError("already playing a trajectory")
    if self.robot.collision_detected:
      self.logger.warn("A collision has been detected while last movement. Removing trigger ...")
      self.robot.clear_collision_detected()

    try: self.robot.stop_move()
    except: pass
    self.robot.set_learning_mode(False)

    i = 0
    for s in self.play_next_step():
      self.logger.debug(f"[{self.trajectory.get_name()}] Playing step number {i}: {s.type.label}
({s.get_value()})")
      i += 1
    rospy.sleep(0.1)

  play = play_all

  def play_async(self):
    if self.is_playing(): raise TrajectoryPlayError("already playing a trajectory")
    self.thread = threading.Thread(target=self.play_all, name="Async"+__class__.__name__, daemon=True)

  def play_next_step(self):
    self.play_started = True
```

```
    for step in self.trajectory:
      if not self.play_started and not self.ignore_errors: raise StopIteration("trajectory has been stopped")
      try:
        step.run(self.robot).raise_if_error()
        yield step
      except Exception as e:
        if self.ignore_errors: self.logger.err(e)
        else: raise TrajectoryPlayError(e) from None

  def stop(self):
    if not self.play_started and not self.ignore_errors: raise TrajectoryPlayError("not playing trajectory")
    self.play_started = False
    self.robot.stop_move()

    if self.thread:
      self.thread.join(5)
      if self.thread.is_alive():
        self.logger.warn("async running of trajectory '@', not stopping! Killing...",
self.trajectory.get_name())
      try: self.thread._stop()
      except: pass
      self.thread = None

```

robot\trajectory\step__init__.py

```

```

robot\trajectory\step\executor.py

```
from util.tests import Tests
from util.vars import *

from .value import StepValue
from .goal import GoalReport

import enum
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

__all__ = [
  "StepExecutor",
  "StepExecutors"
]


class StepExecutor:
  def __init__(self, executor: 'function[[NiryoRosWrapper, object], tuple]'):
    self.__executor = executor

  def run(self, robot: NiryoRosWrapper, args: StepValue) -> GoalReport:
    return GoalReport(self.executor, (robot, args.value))

  @property
  def executor(self):
    return self.__executor


class _executor_funcs_temp:
  @staticmethod
  def start_executor(robot, args):
    result = StepExecutors.tool.value.run(robot, args[0])
    result.raise_if_error()
    result = StepExecutors.move.value.run(robot, args[1])
    result.raise_if_error()
    return (result.result.status, result.result.message)


class StepExecutors(enum.Enum):
  move = StepExecutor(lambda robot, args: Tests.list(args) and Tests.object_size(AXIS_JOINTS_COUNT, args) and
all(Tests.float(v) for v in args) and robot.move_joints(*args))
  tool = StepExecutor(lambda robot, args: Tests.bool(args) and (robot.grasp_with_tool() if args else
robot.release_with_tool()))
```

```
  start = StepExecutor(lambda robot, args: Tests.list(args) and Tests.object_size(2, args) and _executor_funcs_temp.start_executor(robot, args))
  acceleration = StepExecutor(lambda robot, args: Tests.int(args) and robot.set_arm_max_velocity(args))
  none = StepExecutor(lambda *_: (0, "Nothing to do"))

```

robot\trajectory\step\goal.py

```
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapperException

#import dataclasses
#
#@dataclasses.dataclass
#class _sevice_response_stub:
#  status: int
#  message: str


class GoalResult:
  def __init__(self, status: int, message: str="", error: Exception=None):
    self.status = status
    self.message = message if message else "<Unknown message>"
    self.error = error


class GoalFailure(Exception, GoalResult):
  def __init__(self, test_result: GoalResult):
    GoalResult.__init__(self, test_result.status, test_result.message, test_result.error)

  def __str__(self):
    return type(self.error).__name__ + ": " + str(self.error) if self.error else "["+str(self.status)+"] " + self.message


class GoalReport(object):
  def __init__(self, function, args, prefix=""):
#    try: result = function(*args) or _sevice_response_stub(status=-1, message="Command never returned status")
#    except GoalFailure as e: self.result = GoalResult(e.status, e.message, e.error)
#    except Exception as e: self.result = GoalResult(-2, "{}{} - {}".format(prefix, " failed" if prefix else "Failed", str(e)), e)
#    else:
#      self.result = GoalResult(result.status)
#      if result.status >= 0: self.result.message = "{}{} - {}".format(prefix,  " succeed" if prefix else "Succeed", result.message)
#      else: self.result.message = "{}{} - {}".format(prefix, " failed" if prefix else "Failed", result.message)

    try: status, message = function(*args) or (-1, "Command never returned status")
    except GoalFailure as e: self.result = GoalResult(e.status, e.message, e.error)
    # TODO: handle stupid niryo error
    #except NiryoRosWrapperException as e:
    #  ...
    except Exception as e: self.result = GoalResult(-2, "{}{} - {}".format(prefix, " failed" if prefix else "Failed", str(e)), e)
    else:
      self.result = GoalResult(status)
      if status >= 0: self.result.message = "{}{} - {}".format(prefix,  " succeed" if prefix else "Succeed", message)
      else: self.result.message = "{}{} - {}".format(prefix, " failed" if prefix else "Failed", message)

  def raise_if_error(self):
    if self.result.status < 0: raise GoalFailure(self.result)



```

robot\trajectory\step\type.py

```
import enum
from util.tests import Tests, Convert
from util.vars import *
from util.errors import StepAliasError, StepDeserializeError
```

```
from .executor import StepExecutor, StepExecutors
from .value import StepValue

__all__ = [
  "StepType",
  "StepTypes",
]


class StepType:
  def __init__(self, id: int, alias: str, label: str, needed: bool, is_property: bool,
               duplicate_allowed: bool, none_value_allowed: bool,
               executor: StepExecutor, serializer: 'function[[object], str]', deserializer: 'function[[str], object]'
  ):
    self.id = id
    if len(alias) != 1: raise TypeError("'alias' must be one char")
    self.alias = alias
    self.label = label
    self.needed = needed
    self.is_property = is_property
    self.duplicate_allowed = duplicate_allowed
    self.none_value_allowed = none_value_allowed
    self.executor = executor
    self.__serializer = serializer
    self.__deserializer = deserializer

  def __repr__(self):
    return f"{__class__.__name__}(id={self.id}, alias={self.alias}, label={repr(self.label)})"

  __str__ = __repr__

  @property
  def serializer(self):
    return self.__serializer

  @property
  def deserializer(self):
    return self.__deserializer

  def serialize(self, value: StepValue) -> str:
    if self.none_value_allowed and value.is_none: return self.alias
    Tests.none(value.value)
    buffer = self.serializer(value.value)
    if not Tests.buff(buffer, raise_err=False): raise TypeError("returned buffer must be str")
    return self.alias + ' ' + buffer.strip()

  def deserialize(self, buffer: str) -> StepValue:
    self.check_buffer(buffer)
    if self.none_value_allowed and self.is_valid_alias(buffer.strip()): return StepValue()
    try: return StepValue(self.deserializer(buffer[len(self.alias)+1:].strip()))
    except Exception as e: raise StepDeserializeError(e)

  def is_valid_alias(self, alias: str) -> bool:
    return self.alias == alias

  def check_buffer(self, buffer: str, raise_err=True) -> bool:
    valid = Tests.buff(buffer, raise_err=raise_err)
    if not valid: return False
    if not buffer: raise ValueError("buffer is empty")
    buffer += ' '
    if buffer[1] != ' ' and buffer[1] != '\t':
      if raise_err: raise StepAliasError("unable to find alias")
      return False
    if not self.is_valid_alias(buffer[0]):
      if raise_err: raise StepAliasError(f"alias is invalid, got {buffer[0]!r}; must be {self.alias!r}")
      return False
    return True


class _step_funcs_temp:
```

```
  @staticmethod
  def serialize_int(values):
    Tests.int(values)
    return str(values)

  @staticmethod
  def deserialize_int(buffer):
    return int(buffer)

  @staticmethod
  def serialize_str(values):
    Tests.str(values)
    return values

  @staticmethod
  def deserialize_str(buffer):
    return str(buffer)

  @staticmethod
  def start_serializer(values):
    Tests.list(values)
    Tests.object_size(2, values)
    if any(not isinstance(v, StepValue) for v in values): raise ValueError(f"list must only contains {StepValue.__name__!r} objects")
    return _step_funcs_temp.tool_serializer(values[0].value) + ' ' + _step_funcs_temp.move_serializer(values[1].value)

  @staticmethod
  def start_deserializer(buffer):
    i = buffer.find(' ')
    if i < 1: raise ValueError("unable to decode values")
    return [StepValue(_step_funcs_temp.tool_deserializer(buffer[0:i])), StepValue(_step_funcs_temp.move_deserializer(buffer[i:]))]

  @staticmethod
  def move_serializer(values):
    Tests.list(values)
    Tests.object_size(AXIS_JOINTS_COUNT, values)
    Tests.float(*values)
    return ' '.join(str(round(v, AXIS_JOINTS_PRECISION)) for v in values)

  @staticmethod
  def move_deserializer(buffer):
    out = [float(v) for v in buffer.split(' ') if v]
    Tests.object_size(AXIS_JOINTS_COUNT, out)
    return out

  @staticmethod
  def tool_serializer(values):
    Tests.bool(values)
    return str(int(values))

  @staticmethod
  def tool_deserializer(buffer):
    return Convert.to_bool(buffer)


class StepTypes(enum.Enum):
  label = StepType(0, 'l', "label", True, True, False, False,
      StepExecutors.none.value, _step_funcs_temp.serialize_str, _step_funcs_temp.deserialize_str)
  id =  StepType(1, 'i', "id", True, True, False, False,
      StepExecutors.none.value, _step_funcs_temp.serialize_int, _step_funcs_temp.deserialize_int)
  start = StepType(2, 's', "start point", True, True, False, False,
      StepExecutors.start.value, _step_funcs_temp.start_serializer, _step_funcs_temp.start_deserializer)
  move = StepType(3, 'm', "movement", False, False, True, False,
      StepExecutors.move.value, _step_funcs_temp.move_serializer, _step_funcs_temp.move_deserializer)
  tool = StepType(4, 't', "tool", False, False, True, False,
      StepExecutors.tool.value, _step_funcs_temp.tool_serializer, _step_funcs_temp.tool_deserializer)
  acceleration = StepType(5, 'a', "motor acceleration", False, False, True, False,
      StepExecutors.acceleration.value, _step_funcs_temp.serialize_int, _step_funcs_temp.deserialize_int)
  file = StepType(6, 'f', "file name", False, True, False, False,
```

```
      StepExecutors.none.value, _step_funcs_temp.serialize_str, _step_funcs_temp.deserialize_str)
  none = StepType(255, 'u', "<unknown>", False, False, True, True,
      StepExecutors.none.value, lambda *_: '', lambda *_: None)

  @staticmethod
  def find(buffer: str) -> StepType:
    for type in StepTypes:
      if type.value.check_buffer(buffer, raise_err=False): return type.value
    return StepTypes.none.value

  @staticmethod
  def select(consumer: 'function[[StepType], bool]') -> 'list[StepType]':
    l = []
    for type in StepTypes:
      if consumer(type.value): l.append(type.value)
    return l

  @staticmethod
  def get(consumer: 'function[[StepType], bool]') -> StepType:
    for type in StepTypes:
      if consumer(type.value): return type.value
    return StepTypes.none.value


TRAJECTORY_BANNED_DUPLICATED_STEPS.extend(StepTypes.select(lambda type: not type.duplicate_allowed))
TRAJECTORY_NEEDED_STEPS.extend(StepTypes.select(lambda step: step.needed))
TRAJECTORY_PROPERTY_STEPS.extend(StepTypes.select(lambda step: step.is_property))
```

robot\trajectory\step\value.py

```
class StepValue:
  def __init__(self, value: object=None):
    self.__value = value

  def __repr__(self) -> str:
    return f"{__class__.__name__}({repr(self.value)})"

  __str__ = __repr__

  @property
  def is_none(self) -> bool:
    return self.value is None

  @property
  def value(self) -> object:
    return self.__value


```

robot\trajectory\trajectory.py

```
from .step.type import *
from .step.value import *
from .controller import TrajectoryStep, PropertyStep, TrajectoryController

from util.vars import TRAJECTORY_MAX_ID, TRAJECTORY_IDS

__all__ = [
  "Trajectory"
]


class Trajectory(TrajectoryController):
  @staticmethod
  def clear_all_ids():
    TRAJECTORY_IDS.clear()

  def new_id(self) -> TrajectoryController:
    size = len(TRAJECTORY_IDS)
    if size > TRAJECTORY_MAX_ID: raise ValueError("cannot assign new id, limit reached")
    self.set_id(0 if size == 0 else max(TRAJECTORY_IDS)+1)
```

```
    return self

  def set_property(self, step: PropertyStep) -> TrajectoryController:
    """Intercept .set_property() to save ids and avoid to duplicate it"""
    if step.type == StepTypes.id.value:
      new = step.get_value()
      if new in TRAJECTORY_IDS: raise ValueError(f"duplicated trajectory id {new!r}, use .new_id()")
      old = self.get_id()
      if old is not None: TRAJECTORY_IDS.remove(old)
      TRAJECTORY_IDS.append(new)
    return super().set_property(step)

  def set_start_state(self, tool: bool, joints: 'tuple[float]') -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.start.value, StepValue((StepValue(tool), StepValue(joints)))))
    return self

  def get_start_state(self) -> 'tuple[bool, tuple[float]]':
    result = self.get_property(StepTypes.start.value)
    return None if result is None else [v.value if isinstance(v, StepValue) else v for v in
result.get_value()]

  def set_name(self, name: str) -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.label.value, StepValue(name)))
    return self

  def get_name(self) -> str:
    result = self.get_property(StepTypes.label.value)
    return None if result is None else result.get_value()

  def set_id(self, id: int) -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.id.value, StepValue(id)))
    return self

  def get_id(self) -> int:
    result = self.get_property(StepTypes.id.value)
    return None if result is None else result.get_value()

  def set_filename(self, filename: str) -> 'Trajectory':
    self.set_property(TrajectoryStep(StepTypes.file.value, StepValue(filename)))
    return self

  def get_filename(self) -> str:
    result = self.get_property(StepTypes.file.value)
    return None if result is None else result.get_value()

  def add_joints(self, joints: 'tuple[float]') -> 'Trajectory':
    return self.add_step(TrajectoryStep(StepTypes.move.value, StepValue(joints)))

  def add_tool_state(self, state: bool) -> 'Trajectory':
    return self.add_step(TrajectoryStep(StepTypes.tool.value, StepValue(state)))

  def add_motor_speed(self, percent: int) -> 'Trajectory':
    return self.add_step(TrajectoryStep(StepTypes.acceleration.value, StepValue(percent)))

```

robot\util__init__.py

```

```

robot\util\buttons__init__.py

```
from .save_button import *
from .free_motion_button import *


del save_button, free_motion_button
```

robot\util\buttons\free_motion_button.py

```
from threading import Lock, Event
import rospy

```

```python
from niryo_robot_python_ros_wrapper.ros_wrapper_enums import ButtonAction

from end_effector_interface.msg import EEButtonStatus

from niryo_robot_msgs.msg import CommandStatus

__all__ = [
    "FreeMotionButtonRosWrapper",
    "FreeMotionButtonRosWrapperException"
]


class FreeMotionButtonRosWrapperException(Exception):
    pass


class FreeMotionButtonRosWrapper:
    def __init__(self, hardware_version='ned2'):
        self.__hardware_version = hardware_version

        self.__action_lock = Lock()
        self.__action_events = {
            ButtonAction.HANDLE_HELD_ACTION: Event(),
            ButtonAction.LONG_PUSH_ACTION: Event(),
            ButtonAction.SINGLE_PUSH_ACTION: Event(),
            ButtonAction.DOUBLE_PUSH_ACTION: Event(),
            ButtonAction.NO_ACTION: Event(),
        }

        self.__free_motion_button_state = EEButtonStatus.NO_ACTION

        self.__free_motion_button_topic = rospy.Subscriber(
            '/niryo_robot_hardware_interface/end_effector_interface/free_drive_button_status',
            EEButtonStatus, self.__callback_save_pos_button_status)

        self.callback = lambda button_status: None

    def __check_ned_2_version(self):
        if self.__hardware_version != 'ned2':
            raise FreeMotionButtonRosWrapperException(
                "Error Code : {}\nMessage : Wrong robot hardware version, feature only available on Ned2"
.format(
                    CommandStatus.BAD_HARDWARE_VERSION))

    @property
    def hardware_version(self):
        return self.__hardware_version

    @property
    def state(self):
        """
        Get the button state from the ButtonAction class

        :return: int value from the ButtonAction class
        :rtype: int
        """
        self.__check_ned_2_version()
        return self.__free_motion_button_state

    def is_pressed(self):
        """
        Button press state

        :rtype: bool
        """
        self.__check_ned_2_version()
        return self.__free_motion_button_state != EEButtonStatus.NO_ACTION

    def wait_for_action(self, action, timeout=0):
        """
        Waits until a specific action occurs and returns true. Returns false if the timeout is reached.
```

```
74
75          :param action: int value from the ButtonAction class
76          :type action: int self.__save_button_topic
77          :type timeout: float ref_count
78          :return: Returns the detected action, or ButtonAction.NO_ACTION if the timeout is reached  without
    any action.
79          :rtype: int
80          """
81          self.__check_ned_2_version()
82          return self.__wait_any(timeout)
83
84      def wait_for_any_action(self, timeout=0):
85          """
86          Returns the detected action. Returns ButtonAction.NO_ACTION if the timeout is reached  without
    action.
87
88          :type timeout: float
89          :return: Returns the detected action, or ButtonAction.NO_ACTION if the timeout is reached  without
    any action.
90          :rtype: int
91          """
92          self.__check_ned_2_version()
93          return self.__wait_any(timeout)
94
95      def get_and_wait_press_duration(self, timeout=0):
96          """
97          Waits for the button to be pressed and returns the press time.
98          Returns 0 if no press is detected after the timeout duration.
99
100         :type timeout: foat
101         :rtype: float
102         """
103         self.__check_ned_2_version()
104         return self.__get_press_time(timeout)
105
106     def __clear(self):
107         with self.__action_lock:
108             for a_event in self.__action_events.values():
109                 a_event.clear()
110
111     def __set(self, action):
112         self.__free_motion_button_state = action
113         if action in self.__action_events:
114             with self.__action_lock:
115                 self.__action_events[action].set()
116
117     def __wait(self, action, timeout=0.0):
118         if action not in self.__action_events:
119             return False
120
121         self.__action_events[action].clear()
122
123         start_time = rospy.Time.now()
124         while not self.__action_events[action].is_set() and not rospy.is_shutdown():
125             self.__action_events[action].wait(0.1)
126
127             if self.__action_events[action].is_set():
128                 return True
129             elif 0 < timeout < (rospy.Time.now() - start_time).to_sec():
130                 return False
131
132         if self.__action_events[action].is_set():
133             return True
134
135         return False
136
137     def __wait_any(self, timeout=0.0):
138         start_time = rospy.Time.now()
139         self.__clear()
140         if not self.__wait(ButtonAction.HANDLE_HELD_ACTION, timeout=timeout):
141             return ButtonAction.NO_ACTION
```

```
142
143         while not rospy.is_shutdown():
144             if 0 < timeout < (rospy.Time.now() - start_time).to_sec():
145                 break
146
147             for action_name in self.__action_events:
148                 if action_name not in [ButtonAction.NO_ACTION, ButtonAction.HANDLE_HELD_ACTION] \
149                         and self.__action_events[action_name].is_set():
150                     return action_name
151
152             rospy.sleep(0.1)
153
154         return ButtonAction.HANDLE_HELD_ACTION
155
156     def __get_press_time(self, timeout=0.0):
157         if not self.__wait(ButtonAction.HANDLE_HELD_ACTION, timeout=timeout):
158             return 0
159
160         pressed_time = rospy.Time.now()
161
162         if not self.__wait(ButtonAction.NO_ACTION):
163             return 0
164
165         return (rospy.Time.now() - pressed_time).to_sec()
166
167     def __callback_save_pos_button_status(self, free_motion_button_status):
168         self.__set(free_motion_button_status.action)
169         self.callback(free_motion_button_status)
170
```

robot\util\buttons\save_button.py

```
1  from threading import Lock, Event
2  import rospy
3
4  from niryo_robot_python_ros_wrapper.ros_wrapper_enums import ButtonAction
5
6  from end_effector_interface.msg import EEButtonStatus
7
8  from niryo_robot_msgs.msg import CommandStatus
9
10 __all__ = [
11     "SaveButtonRosWrapper",
12     "SaveButtonRosWrapperException"
13 ]
14
15
16 class SaveButtonRosWrapperException(Exception):
17     pass
18
19
20 class SaveButtonRosWrapper:
21     def __init__(self, hardware_version='ned2'):
22         self.__hardware_version = hardware_version
23
24         self.__action_lock = Lock()
25         self.__action_events = {
26             ButtonAction.HANDLE_HELD_ACTION: Event(),
27             ButtonAction.LONG_PUSH_ACTION: Event(),
28             ButtonAction.SINGLE_PUSH_ACTION: Event(),
29             ButtonAction.DOUBLE_PUSH_ACTION: Event(),
30             ButtonAction.NO_ACTION: Event(),
31         }
32
33         self.__save_button_state = EEButtonStatus.NO_ACTION
34
35         self.__save_button_topic = rospy.Subscriber(
36             '/niryo_robot_hardware_interface/end_effector_interface/save_pos_button_status',
37             EEButtonStatus, self.__callback_save_pos_button_status)
38
39
```

```
        self.callback = lambda button_status: None


    def __check_ned_2_version(self):
        if self.__hardware_version != 'ned2':
            raise SaveButtonRosWrapperException(
                "Error Code : {}\nMessage : Wrong robot hardware version, feature only available on Ned2"
.format(
                    CommandStatus.BAD_HARDWARE_VERSION))

    @property
    def hardware_version(self):
        return self.__hardware_version

    @property
    def state(self):
        """
        Get the button state from the ButtonAction class

        :return: int value from the ButtonAction class
        :rtype: int
        """
        self.__check_ned_2_version()
        return self.__save_button_state

    def is_pressed(self):
        """
        Button press state

        :rtype: bool
        """
        self.__check_ned_2_version()
        return self.__save_button_state != EEButtonStatus.NO_ACTION

    def wait_for_action(self, action, timeout=0):
        """
        Waits until a specific action occurs and returns true. Returns false if the timeout is reached.

        :param action: int value from the ButtonAction class
        :type action: int
        :type timeout: float
        :return: True if the action has occurred, false otherwise
        :rtype: bool
        """
        self.__check_ned_2_version()
        return self.__wait(action, timeout)

    def wait_for_any_action(self, timeout=0):
        """
        Returns the detected action. Returns ButtonAction.NO_ACTION if the timeout is reached  without
action.

        :type timeout: float
        :return: Returns the detected action, or ButtonAction.NO_ACTION if the timeout is reached  without
any action.
        :rtype: int
        """
        self.__check_ned_2_version()
        return self.__wait_any(timeout)

    def get_and_wait_press_duration(self, timeout=0):
        """
        Waits for the button to be pressed and returns the press time.
        Returns 0 if no press is detected after the timeout duration.

        :type timeout: foat
        :rtype: float
        """
        self.__check_ned_2_version()
        return self.__get_press_time(timeout)

```

```
    def __clear(self):
        with self.__action_lock:
            for a_event in self.__action_events.values():
                a_event.clear()

    def __set(self, action):
        self.__save_button_state = action
        if action in self.__action_events:
            with self.__action_lock:
                self.__action_events[action].set()

    def __wait(self, action, timeout=0.0):
        if action not in self.__action_events:
            return False

        self.__action_events[action].clear()

        start_time = rospy.Time.now()
        while not self.__action_events[action].is_set() and not rospy.is_shutdown():
            self.__action_events[action].wait(0.1)

            if self.__action_events[action].is_set():
                return True
            elif 0 < timeout < (rospy.Time.now() - start_time).to_sec():
                return False

        if self.__action_events[action].is_set():
            return True

        return False

    def __wait_any(self, timeout=0.0):
        start_time = rospy.Time.now()
        self.__clear()
        if not self.__wait(ButtonAction.HANDLE_HELD_ACTION, timeout=timeout):
            return ButtonAction.NO_ACTION

        while not rospy.is_shutdown():
            if 0 < timeout < (rospy.Time.now() - start_time).to_sec():
                break

            for action_name in self.__action_events:
                if action_name not in [ButtonAction.NO_ACTION, ButtonAction.HANDLE_HELD_ACTION] \
                        and self.__action_events[action_name].is_set():
                    return action_name

            rospy.sleep(0.1)

        return ButtonAction.HANDLE_HELD_ACTION

    def __get_press_time(self, timeout=0.0):
        if not self.__wait(ButtonAction.HANDLE_HELD_ACTION, timeout=timeout):
            return 0

        pressed_time = rospy.Time.now()

        if not self.__wait(ButtonAction.NO_ACTION):
            return 0

        return (rospy.Time.now() - pressed_time).to_sec()

    def __callback_save_pos_button_status(self, save_button_status):
        self.__set(save_button_status.action)
        self.callback(save_button_status)

```

robot\util\cli.py

```
"""Command line parser for project"""

import os
```

```python
from argparse import ArgumentParser
from util.logger import Log, LogLevel

__all__ = ["CliParser"]


class CliParser(ArgumentParser):
  def __init__(self):
    super().__init__(usage="%(prog)s [OPTIONS]",
                     epilog="Note: Logs writing will be done into files, split every 512 KB.")

    # Create arguments
    self.add_argument("--no-calibration",
                      action="store_true",
                      help="disable auto calibration at start of program",
                      dest="nocalibration")
    self.add_argument("--no-trajectories",
                      action="store_true",
                      help="do not load trajectories",
                      dest="notrajectories")
    self.add_argument("--no-modbus",
                      action="store_true",
                      help="disable Modbus/TCP server",
                      dest="nomodbus")
    self.add_argument("--no-terminal",
                      action="store_true",
                      help="disable the terminal",
                      dest="noterminal")
    self.add_argument("--no-status-sound",
                      action="store_true",
                      help="disable colors when logging",
                      dest="nostatussound")
    self.add_argument("--no-default-pose",
                      action="store_true",
                      help="do not move to default pose when stopping robot",
                      dest="nodefaultpose")
    self.add_argument("--no-colors",
                      action="store_true",
                      help="disable colors when logging",
                      dest="nocolors")
    self.add_argument("--no-write-logs",
                      action="store_true",
                      help="disable writting of logs in files",
                      dest="nowritelogs")
    self.add_argument("--logs-path",
                      action="store",
                      default=os.getcwd(),
                      type=str,
                      help="the directory to write logs. (default is working directory)",
                      dest="logspath")
    self.add_argument("--log-folder",
                      action="store",
                      default=Log.folder_name,
                      type=str,
                      help="the folder name to store log files. (default is 'logs')",
                      dest="name")
    self.add_argument("--log-level",
                      action="store",
                      default="info",
                      choices=[l.name for l in LogLevel],
                      help="set a default logging level",
                      dest="loglevel")

```

robot\util\colors.py

```python
"""Library to easily normalise colors into RGB list"""

from .tests import Tests

class Colors:
```

```
  """https://github.com/numworks/epsilon/blob/master/kandinsky/include/kandinsky/color.h"""

  blue    = [0,   0,   248]
  b       = blue
  red     = [248, 0,   0]
  r       = red
  green   = [80,  192, 0]
  g       = green
  yellow  = [248, 252, 0]
  y       = yellow
  brown   = [136, 112, 80]
  black   = [0,   0,   0]
  k       = black
  white   = [248, 252, 248]
  w       = white
  pink    = [248, 168, 176]
  orange  = [248, 132, 24]
  purple  = [104, 44,  120]
  grey    = [160, 164, 160]
  gray    = grey
  cyan    = [0,   252, 248]
  magenta = [248, 4,   136]
  COLORS = {k: v for k, v in locals().items() if not k.startswith('_')}

  fix = lambda r, g, b, expand=0: (
    (Colors.expand((r>>3)&0x1f, 5) if expand else ((r>>3)&0x1f)<<3),
    (Colors.expand((g>>2)&0x3f, 6) if expand else ((g>>2)&0x3f)<<2),
    (Colors.expand((b>>3)&0x1f, 5) if expand else ((b>>3)&0x1f)<<3))
  fix2 = lambda *values, expand=0: (
    tuple(Colors.expand(Colors.contract(c, i%2, 3), 5+i%2) for i, c in enumerate(*values))
      if expand else
    tuple(Colors.contract(c, i%2, 3)<<(3-i%2) for i, c in enumerate(*values)))
  contract = lambda v, nBits, length=8: (v>>(length-nBits))&(2**(4+length-nBits)-1)
  expand = lambda v, nBits, length=8: (v<<(length-nBits))|(v>>(nBits-(length-nBits)))

  def convert(rgbOrName):
    _type = type(rgbOrName)

    if _type == str:
      if rgbOrName in Colors.COLORS: return Colors.fix(*Colors.COLORS[rgbOrName])
      elif rgbOrName.startswith('#'):
        if len(rgbOrName) != 7: raise ValueError("RGB hex values are 6 bytes long")

        try: return Colors.fix(*[int(rgbOrName[i:i+2], 16) for i in range(1, len(rgbOrName), 2)])
        except ValueError as e:
          e.args = (f"invalid literal for int() with base 16: '{rgbOrName[1:]}'",)
          raise
      else:
        try: ratio = float(rgbOrName)
        except ValueError as e:
          e.args = ("invalid syntax for number",)
          raise
        else:
          if 0 <= ratio <= 1: return tuple([int(255*ratio) for _ in range(3)])
          else: raise ValueError("Gray levels are between 0.0 and 1.0")

    elif _type == int: raise ValueError("Int are not colors")

    Tests.list(rgbOrName)
    if len(rgbOrName) != 3: raise ValueError("Color needs 3 components")
    return Colors.fix(*[int(c) for c in rgbOrName if type(c) == float or Tests.int(c)])

```

robot\util\commands.py

```
  """Commands for the shell"""

  from util.logger import Logger, Log, LogLevel
  from util.errors import TrajectoryNotFoundError
  from util.strings import getchar, lJustList, timedelta_format
  from util.tests import Convert
```

```python
from util.vars import JOINTS_MOVE_OFFSET
from calibration import new_calibration
from trajectory.manager import TRAJECTORY_MANAGER
from trajectory.player import TrajectoryPlayer

from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper
from time import monotonic


class Command:
  def __init__(self, name: str, description: str, command: 'function[[NiryoRosWrapper, Logger], int]', fargs: str="", args_needed: int=0):
    self.name = name
    self.fargs = fargs
    self.description = description
    self.command = command
    self.args_needed = args_needed

  def __call__(self, robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    return self.command(robot, logger, *args)


class ShellCommands:
  def help(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    logger.info("\nNiryo Robot CLI")
    logger.info("\ncommands:")

    best1, best2, n, a, d = 0, 0, [], [], []
    for c in ShellCommands.commands:
      n.append(c.name)
      a.append(c.fargs)
      d.append(c.description)
      if len(c.name) > best1: best1 = len(c.name)
      if len(c.fargs) > best2: best2 = len(c.fargs)

    for n, a, d in zip(lJustList(n, best1+1), lJustList(a, best2+2), d):
      logger.info("  " + n + a + d)

  def loglevel(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    if not args:
      logger.info("Actual level: " + Log.level.name)
      logger.info("Available levels: " + ", ".join(level.name for level in LogLevel))
      return

    for level in LogLevel:
      if level.name == args[0]:
        Log.level = level
        logger.info(f"Log level set to '{level.name}'")
        break
    else: logger.err(f"Level '{args[0]}' not found")

  def manual(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    logger.info("Manual mode started!")
    logger.info("Robot can now be controlled by keyboard.")
    logger.info()
    logger.info("Press &lbENTER&fr to exit this mode.")
    logger.info("Available move keys:")
    for i, v in enumerate((('a','q'), ('z','s'), ('e','d'), ('r','f'), ('t','g'), ('y','h'))):
      logger.info(f"| joint {i+1}: &lg{v[0]}&fr (+) / &lg{v[1]}&fr (-)")
    logger.info("| tool: &lgTAB")

    tool_state = False
    key = ''
    while key != '\r':
      key = getchar()
      logger.debug(f"Received key {key!r}")

      # TODO: optimize that
      last_pos = robot.get_joints()
      if key == 'a':
        last_pos[0] += JOINTS_MOVE_OFFSET
```

```
        robot.move_joints(*last_pos)
      elif key == 'q':
        last_pos[0] -= JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'z':
        last_pos[1] += JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 's':
        last_pos[1] -= JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'e':
        last_pos[2] += JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'd':
        last_pos[2] -= JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'r':
        last_pos[3] += JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'f':
        last_pos[3] -= JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 't':
        last_pos[4] += JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'g':
        last_pos[4] -= JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'y':
        last_pos[5] += JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == 'h':
        last_pos[5] -= JOINTS_MOVE_OFFSET
        robot.move_joints(*last_pos)
      elif key == '\t': # invert tool state
        tool_state = not tool_state
        (robot.grasp_with_tool if tool_state else robot.release_with_tool)()

    logger.info("\nManual mode exited!")

  def deep(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    if not len(args):
      logger.info("Deep learning mode is @", "enabled" if robot.get_learning_mode() else "disabled")
      return
    enabled = Convert.to_bool(args[0])
    robot.set_learning_mode(enabled)
    logger.info("Deep learning mode @", "enabled" if enabled else "disabled")

  def default(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    logger.info("Moving to the default position ...")
    robot.move_to_sleep_pose()

  def halt(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    force_deep = len(args) and (Convert.to_bool(args[0]) or args[0] == 'deep')
    if not force_deep:
      try: robot.stop_move()
      except:
        logger.warn("An error occurred. Trying to stop with deep mode ...")
        force_deep = True
      else: logger.info("Stopped movement")
    if force_deep:
      robot.set_learning_mode(True)
      robot.set_learning_mode(False)
      logger.info("Movement stopped with deep learning mode")

  def clear(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    robot.clear_collision_detected()
    logger.info("Removed collision detected trigger")

  def calibrate(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
```

```
    logger.info("Started calibration.")
    start = monotonic()
    new_calibration(robot)
    logger.info("Calibration finished in @", timedelta_format((monotonic()-start)*1000))

  def play(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    if not args:
      logger.info("Trajectories: ")
      for t in TRAJECTORY_MANAGER:
        logger.info("| Name: @ \tID: @ \tSteps: @", t.get_name(), t.get_id(), t.count_steps())

    elif args[0] == "reload":
      TRAJECTORY_MANAGER.reload()
      logger.info("Trajectories reloaded.")
    else:
      try: TrajectoryPlayer.play_trajectory(robot, args[0])
      except TrajectoryNotFoundError as e: logger.err(str(e))

  def exec_(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    logger.info("Running script ...")
    start = monotonic()
    exec(" ".join(args), globals(), locals())
    logger.info("Script finished in @", timedelta_format((monotonic()-start)*1000))

  def speed(robot: NiryoRosWrapper, logger: Logger, *args: 'tuple[str]'):
    if not args: logger.info("Arm max velocity is at @%", robot.get_max_velocity_scaling_factor())
    else:
      v = int(args[0])
      robot.set_arm_max_velocity(v)
      logger.info("Arm max velocity set to @%", v)

  commands: 'list[Command]'  = [
    Command("help", "Print this help", help),
    Command("exit", "Exit the main program", lambda *_: 255),
    Command("loglevel", "Control verbosity level of logging", loglevel, "[level]"),
    Command("manual", "Control the robot manually by keyboard", manual),
    Command("deep", "Set/Get deep learning mode", deep, "[on|off]"),
    Command("default", "Move robot to default position", default),
    Command("halt", "Stop the current robot movement", halt, "[deep]"),
    Command("clear", "Clear collision detected trigger", clear),
    Command("calibrate", "Start new robot calibration", calibrate),
    Command("play", "Play a trajectory or list it", play, "[reload|type-<name|id>]"),
    Command("exec", "Execute python code in runtime", exec_, "<code...>", 1),
    Command("speed", "Set/Get arm max speed (velocity) between 1 and 200%", speed, "[percent]")
  ]


```

robot\util\errors.py

```
"""All errors classes here"""

class StepAliasError(Exception): ...

class StepDeserializeError(Exception):
  def __init__(self, exc: Exception, *args: object):
    super().__init__(exc, *args)
    self.exc = exc

  def __str__(self) -> str:
    t = type(self.exc)
    return t.__name__ + ": " + super().__str__() if issubclass(t, Exception) else str(self.exc)

class TrajectoryReadError(Exception):
  def __init__(self, file: str, message: str, *args: object):
    super().__init__(*args)
    self.file = file
    self.message = message

  def __str__(self) -> str:
    m = super().__str__()
```

```
    return f"[{self.file}] {self.message}" + (" :" + m if m and m.strip() != ':' else '')

class TrajectoryLearningError(Exception): ...

class TrajectoryPlayError(Exception):
  def __init__(self, exc: Exception, *args: object):
    super().__init__(exc, *args)
    self.exc = exc

  def __str__(self) -> str:
    t = type(self.exc)
    return t.__name__ + ": " + super().__str__() if issubclass(t, Exception) else str(self.exc)

class TrajectoryManagerError(Exception): ...

class TrajectoryNotFoundError(Exception): ...

class ModbusServerError(Exception): ...

```

robot\util\lock.py

```
"""Simple lock file library"""

from util.strings import get_parent_path
import os


class LockFile:
  def __init__(self, file: str) -> None:
    self.__lock_file = file

  @property
  def lockfile(self):
    return self.__lock_file

  @property
  def present(self):
    return os.path.exists(self.lockfile)

  def acquire(self, who: object):
    """Acquire the lock file. 'who' is an object no matter what, but will represent the handler by an id"""
    parent = get_parent_path(self.lockfile)
    if not os.path.exists(parent): os.mkdir(parent)
    with open(self.lockfile, 'wt') as f: f.write(str(id(who)))

  def notify_release(self):
    """Notify that the lock file will be released soon"""
    if not self.present: return
    with open(self.lockfile, 'wt') as f: f.write("0")

  def release(self):
    """Remove the lock file"""
    if self.present: os.remove(self.lockfile)

  def handler(self) -> int:
    """Return the handler id, or 0 if no handler, else -1 if handler is invalid"""
    if not self.present: return 0
    with open(self.lockfile, "rt") as f:
      try: return int(f.read().strip())
      except ValueError: return -1

  def handled(self) -> bool:
    """Return if the lock file has an handler"""
    return self.handler() > 0

```

robot\util\logger.py

```
"""
Little library to log things easily with topics, colors, error wrapping, etc.
```

```
With writing of logs into files and many others features.
"""

import enum, datetime, os, sys, traceback, typing

class Logger:
  input_prompt: str = "&fb&lg>>> &fr"
  log_prompt: str = "&fb&lk[{time}] &lw[{topic}] "

  def __init__(self, topic: str):
    self.topic = topic

  def __call__(self, text: str='', *args: tuple):
    """If logger is called like a function, just print an info"""
    self.info(text, *args)

  def _format_log_prompt(self) -> str:
    return self.log_prompt.format(time=datetime.datetime.now().strftime("%m-%d-%Y %H:%M:%S"), topic=
self.topic)

  def debug(self, text: str='', *args: tuple):
    """
    Log an debug information
    """
    Log.debug(text, *args, pre_tag=self._format_log_prompt())

  def info(self, text: str='', *args: tuple):
    """
    Log an information
    """
    Log.info(text, *args, pre_tag=self._format_log_prompt())

  def warn(self, text: str='', *args: tuple):
    """
    Log an warning
    """
    Log.warn(text, *args, pre_tag=self._format_log_prompt())

  def err(self, text: typing.Union[str, BaseException], *args: tuple, no_stacktrace: bool=False):
    """
    Log an error
    """
    Log.err(text, *args, pre_tag=self._format_log_prompt(), no_stacktrace=no_stacktrace)

  def input(self, prompt: str='', no_block: bool=False) -> str:
    """
    Get an input line
    'no_block' to wait for line to be finished or not
    """
    prompt = (prompt + ' ' if prompt else '') + self.input_prompt
    p = ColorCodes.parse(prompt, Log.use_colors)
    line = (print(p) or sys.stdin.readline()[:-1]) if no_block else input(p)

    # write the input in log files
    if line: Log.write(ColorCodes.parse(prompt, False)+line)

    return line


class ColorCodes:
  """
  Source: https://github.com/Anuken/Arc/blob/master/arc-core/src/arc/util/ColorCodes.java
  """

  flush: str = "\033[H\033[2J"
  reset: str = "\u001B[0m"
  bold: str = "\u001B[1m"
  italic: str = "\u001B[3m"
  underline: str = "\u001B[4m"
  black: str = "\u001B[30m"
  red: str = "\u001B[31m"
```

```
  green: str = "\u001B[32m"
  yellow: str = "\u001B[33m"
  blue: str = "\u001B[34m"
  purple: str = "\u001B[35m"
  cyan: str = "\u001B[36m"
  lightBlack: str = "\u001b[90m"
  lightRed: str = "\u001B[91m"
  lightGreen: str = "\u001B[92m"
  lightYellow: str = "\u001B[93m"
  lightBlue: str = "\u001B[94m"
  lightMagenta: str = "\u001B[95m"
  lightCyan: str = "\u001B[96m"
  lightWhite: str = "\u001b[97m"
  white: str = "\u001B[37m"
  backDefault: str = "\u001B[49m"
  backRed: str = "\u001B[41m"
  backGreen: str = "\u001B[42m"
  backYellow: str = "\u001B[43m"
  backBlue: str = "\u001B[44m"

  if os.name == "nt":
    # Try to enable the VT100 feature, ANSI-like escapes codes
    # Sources:
    #   - https://github.com/Textualize/rich/blob/master/rich/_windows.py
    #   - https://stackoverflow.com/a/54291292
    #   - https://learn.microsoft.com/en-us/windows/console/setconsolemode
    #   - https://stackoverflow.com/a/287944
    import ctypes

    hStdOut = ctypes.windll.kernel32.GetStdHandle(-11)
    mode = ctypes.c_ulong()
    success = ctypes.windll.kernel32.GetConsoleMode(hStdOut, ctypes.byref(mode))

    # if terminal already handle ansi codes, mode.value is 7, so will not enable this
    if not (success and mode.value & 4):
      mode.value |= 4
      ctypes.windll.kernel32.SetConsoleMode(hStdOut, mode)

    # Cleanup
    del ctypes, hStdOut, mode, success

  prefix: str = '&'
  codes: dict = {
    "ff": flush,
    "fr": reset,
    "fb": bold,
    "fi": italic,
    "fu": underline,
    "k":  black,
    "lk": lightBlack,
    "lw": lightWhite,
    "r":  red,
    "g":  green,
    "y":  yellow,
    "b":  blue,
    "p":  purple,
    "c":  cyan,
    "lr": lightRed,
    "lg": lightGreen,
    "ly": lightYellow,
    "lm": lightMagenta,
    "lb": lightBlue,
    "lc": lightCyan,
    "w":  white,
    "bd": backDefault,
    "br": backRed,
    "bg": backGreen,
    "by": backYellow,
    "bb": backBlue
  }
```

```
143
144   @staticmethod
145   def parse(text: str, use_colors: bool=True) -> str:
146     if use_colors: # Parse colors
147       for color in ColorCodes.codes:
148         text = text.replace(f"{ColorCodes.prefix}{color}", ColorCodes.codes[color])
149
150     else: # Remove colors
151       for color in ColorCodes.codes:
152         text = text.replace(f"{ColorCodes.prefix}{color}", "")
153
154     return text
155
156
157 class LogLevel(enum.Enum):
158   debug = 0
159   info = 1
160   warn = 2
161   err = 3
162   none = 4
163
164
165 class Log:
166   """
167   Source: https://github.com/Anuken/Arc/blob/master/arc-core/src/arc/util/Log.java
168   """
169   file_split_size: int = 512 * 1024 # in bytes
170   use_colors: bool = True
171   level: LogLevel = LogLevel.info
172   write_logs: bool = True
173   folder_name: str = "logs"
174   logs_path: str = os.path.join("./", folder_name)
175
176   _current_file: str = None
177
178   @staticmethod
179   def write(text: str):
180     """
181     Source: https://github.com/Anuken/Mindustry/blob/master/server/src/mindustry/server/
ServerControl.java#L1090
182     """
183     if not Log.write_logs: return # not enabled
184
185     # Check if folder is created, else create it
186     if os.path.basename(Log.logs_path) != Log.folder_name: Log.logs_path = os.path.join(Log.logs_path,
Log.folder_name)
187     if not os.path.exists(Log.logs_path): os.mkdir(Log.logs_path)
188
189     # Check if current file size is exeeded
190     if Log._current_file != None and os.path.getsize(Log._current_file) > Log.file_split_size:
191       with open(Log._current_file, 'at', encoding='utf-8') as f:
192         f.write("[End of log file. Date: "+ datetime.datetime.now().strftime("%m-%d-%Y %H:%M:%S") + "]\n")
193       Log._current_file = None
194
195     # Check for new file
196     if Log._current_file == None:
197       i = 0
198       p = os.path.join(Log.logs_path, f"log-{i}.txt")
199
200       while os.path.exists(p) and (os.path.isdir(p) or os.path.getsize(p) > Log.file_split_size):
201         i += 1
202         p = os.path.join(Log.logs_path, f"log-{i}.txt")
203
204       # Found available file, open it now
205       Log._current_file = p
206
207     # And write the content
208     with open(Log._current_file, 'at', encoding='utf-8') as f:
209       f.write(text+'\n')
210
211   @staticmethod
```

```
  def format(text: str, *args: tuple, format_prefix: str='@'):
    if len(args):
      for arg in args: text = text.replace(format_prefix, f"&lb&fb{str(arg)}&fr", 1)
    return text

  @staticmethod
  def log(level: LogLevel, text: str, *args: tuple, pre_tag: str=''):
    if Log.level.value > level.value: return

    text = Log.format(str(text), *args)

    for line in text.split('\n'):
      line = (pre_tag +
        ("&lc&fb[D]&fr " if level == LogLevel.debug else
          "&lb&fb[I]&fr " if level == LogLevel.info else
          "&ly&fb[W]&fr " if level == LogLevel.warn else
          "&fr&lr&fb[E] " if level == LogLevel.err else
          "") +
        line + "&fr")

      print(ColorCodes.parse(line, Log.use_colors))
      Log.write(ColorCodes.parse(line, False))

  @staticmethod
  def debug(text: str='', *args: tuple, pre_tag: str=''):
    Log.log(LogLevel.debug, text, *args, pre_tag=pre_tag)

  @staticmethod
  def info(text: str='', *args: tuple, pre_tag: str=''):
    Log.log(LogLevel.info, text, *args, pre_tag=pre_tag)

  @staticmethod
  def warn(text: str='', *args: tuple, pre_tag: str=''):
    Log.log(LogLevel.warn, text, *args, pre_tag=pre_tag)

  @staticmethod
  def err(text: typing.Union[str, BaseException], *args: tuple, pre_tag: str='', no_stacktrace: bool=False):
    if isinstance(text, BaseException):
      text = ''.join(traceback.format_exception_only(type(text), value=text) if no_stacktrace else traceback.format_exception(type(text), value=text, tb=text.__traceback__))[:-1]
    Log.log(LogLevel.err, text, *args, pre_tag=pre_tag)

```

robot\util\rospy_logger.py

```
import rospy
from .logger import Logger


def redirect_rospy_logger(logger_topic: str="RosPy"):
  logger = Logger(logger_topic)

  def logger_redirector(msg, args, kwargs, throttle=None, throttle_identical=False, level='info', once=False):
    func = getattr(logger, level, None)
    if func is None: func = logger.err if level == 'critical' else logger.info
    if len(args):
      for arg in args: msg = msg.replace("%s", str(arg), 1)
    func(msg, args)

  rospy.core._base_logger = logger_redirector

```

robot\util\shell.py

```
"""Wrapper to execute commands"""

from util.logger import Logger
from util.vars import EXIST_REQUESTED_EVENT
from .commands import ShellCommands

from threading import Thread
```

```python
from niryo_robot_python_ros_wrapper.ros_wrapper import NiryoRosWrapper

__all__ = [
  "Shell",
  "ShellManager"
]


class Shell:
  def __init__(self, robot: NiryoRosWrapper):
    self.robot = robot
    self.logger = Logger(__class__.__name__)

  def parse_command(self, command_args: 'tuple[str]') -> int:
    for command in ShellCommands.commands:
      if command_args[0] == command.name:
        if len(command_args)-1 < command.args_needed:
          self.logger.err(f"The command '{command_args[0]}' need {command.args_needed} argument" + ('s' if command.args_needed > 1 else ''))
          return
        else: return command(self.robot, self.logger, *command_args[1:])
    else: self.logger.err(f"Command '{command_args[0]}' not found. See 'help' for usage")

  def wait_process_command(self) -> int:
    """
    Request a command to user and execute it.

    This, block the input, so run this in another thread

    @ Return 0 if no error.
    @ Return 1 if an error is raised by a command.
    @ Return 127 if empty command
    @ Return 255 if exit requested
    """
    try:
      command_args = tuple([arg for arg in self.logger.input().split(' ') if arg])
      if not len(command_args) or command_args[0] == '': return 127
      return self.parse_command(command_args) or 0

    except EOFError:
      # started with service, there are no stdin, so wait a little and print nothing to avoid filling logs
      import time
      time.sleep(0.1)

    except Exception as e:
      self.logger.warn("Command raised an error during execution.")
      self.logger.err(e, no_stacktrace=True)

      return 1
    return 0


class ShellManager(Thread):
  def __init__(self, robot: NiryoRosWrapper):
    super().__init__(name="Shell", daemon=True)
    self.robot = robot
    self.shell = Shell(self.robot)

  def run(self):
    try:
      self.shell.logger.info("Shell started")

      while not EXIST_REQUESTED_EVENT.is_set():
        if self.shell.wait_process_command() == 255: EXIST_REQUESTED_EVENT.set()
    except BaseException as e:
      self.shell.logger.err("\nError while running shell!\n")
      self.shell.logger.err(e)

    self.shell.logger.info("Shell stopped")

```

robot\util\strings.py

```python
"""String utilities"""

import re, os

filename_pattern = re.compile(r"[\0/\"<>|:*?\\]")
reserved_filename_pattern = re.compile(r"(CON|AUX|PRN|NUL|(COM[0-9])|(LPT[0-9]))((\..*$)|$)", re.IGNORECASE)
time_periods = (
  ('year',   1000*60*60*24*365),
  ('month',  1000*60*60*24*30),
  ('day',    1000*60*60*24),
  ('hour',   1000*60*60),
  ('minute', 1000*60),
  ('second', 1000*1),
  ('milisecond', 1),
)


def sanitize_filename(filename: str) -> str:
  """
  Replaces non-safe filename characters with '_'. Handles reserved window file names.

  Source: https://github.com/Anuken/Arc/blob/master/arc-core/src/arc/util/Strings.java
  """
  if reserved_filename_pattern.match(filename): return "a" + filename
  return ''.join([c for c in filename_pattern.sub("_", filename) if c.isprintable()])

def find_available_file(file: str, start_index: int=0, separator: str='-') -> str:
  """
  Find a non-existing file name, with an index. \n
  The separator and the index will be added if the specified file already exist.

  Accepted formatting:
    * '{sep}': separator between name and index, can be changed with 'separator' parameter
    * '{index}': file index

  (e.g. 'log{sep}{index}.txt')
  """
  f = file.format(sep='', index='')
  # Do a first check before adding index
  if not os.path.exists(f) and not os.path.isdir(f): return f
  # Add default formatting if not added in 'file'
  if file == f: file += "{sep}{index}"
  # Find a non-existing file
  f = file.format(sep=separator, index=start_index)
  while os.path.exists(f) or os.path.isdir(f):
    start_index += 1
    f = file.format(sep=separator, index=start_index)
  return f

def get_parent_path(path: str) -> str:
  """
  Get parent directory or itself if is the root directory.
  """
  if not path: return path
  p = path[:-1] if path.endswith('/') or path.endswith('\\') else path
  p = p[:p.rfind('/')+1 or p.rfind('\\')+1]
  return p if p else path

def get_child_path(path: str) -> str:
  """
  Get last child of a path or itself if is the root directory.
  """
  if not path: return path
  p = path[:-1] if path.endswith('/') or path.endswith('\\') else path
  p = p[p.rfind('/')+1 or p.rfind('\\')+1:]
  return p if p else path

def getchar() -> str:
    """Returns a single character from standard input"""
```

```
    ch = ''
    if os.name == 'nt': # how it works on windows
        import msvcrt
        ch = msvcrt.getch()
    else:
        import tty, termios, sys
        fd = sys.stdin.fileno()
        old_settings = termios.tcgetattr(fd)
        try:
            tty.setraw(sys.stdin.fileno())
            ch = sys.stdin.read(1)
        finally:
            termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    if ord(ch) == 3: raise KeyboardInterrupt # handle ctrl+C
    return ch


def rJust(str: str, newLength: int, filler: str=" ") -> str:
  if len(filler) == 0: return str
  return filler * ((newLength - len(str)) // len(filler)) + filler[:((newLength - len(str)) % len(filler))]
+ str

def rJustList(lst: 'list[str]', newLength: int, filler: str=" ") -> 'list[str]':
  return [rJust(s, newLength, filler)  for s in lst]

def lJust(str: str, newLength: int, filler: str=" ") -> str:
  if len(filler) == 0: return str
  return str + filler * ((newLength - len(str)) // len(filler)) + filler[:((newLength - len(str)) %
len(filler))]

def lJustList(lst: 'list[str]', newLength: int, filler: str=" ") -> 'list[str]':
  return [lJust(s, newLength, filler) for s in lst]

def mJust(left: str, right: str, newLength: int, filler: str=" ") -> str:
  if len(filler) == 0: return left + right
  s = newLength - len(left) - len(right)
  return left + filler * (s // len(filler)) + filler[: (s % len(filler))] + right

def mJustList(left: 'list[str]', right: 'list[str]', newLength: int, filler: str=" ") -> 'list[str]':
  arr = []
  for l, r in zip(left, right): arr.append(mJust(l, r, newLength, filler))
  if len(left) > len(right): arr.extend(lJustList(left[len(right):], newLength, filler))
  elif len(right) > len(left): arr.extend(rJustList(right[len(left):], newLength, filler))
  return arr


def timedelta_format(delta_ms):
  delta_ms = int(delta_ms)
  strings=[]
  for period_name, period_seconds in time_periods:
    if delta_ms > period_seconds:
      period_value, delta_ms = divmod(delta_ms, period_seconds)
      strings.append("%s %s%s" % (period_value, period_name, 's'*(period_value>1)))

  return (strings[0] if len(strings) < 2 else ", ".join(strings[:-1]) + " and " + strings[-1]) if strings
else "0 "+time_periods[-1][0]

```

robot\util\tests.py

```
"""Test type of things"""

__all__ = [
  "Tests",
  "Convert"
]

def _create_test(message, *valid_types, reverse=False):
  """
  Decorator to create the test. Put this before an empty method with (*objects, raise_err=True) arguments.

```

```
  The message can contains {found} and {valid} keys to format found type and required types.

  If reverse is True: checks if type is anything except valid_types.

  """
  if type(message) != str: raise TypeError("message must be an str")
  if len(valid_types) == 0: raise IndexError("one type or more must be specified")
  if not all(issubclass(type(t), type) for t in valid_types): raise TypeError("valid_types must be a list of types")

  def decorator(func):
    type_msg = valid_types[0].__name__ if len(valid_types) == 1 else (", ".join([i.__name__ for i in valid_types[:-1]])+" or "+valid_types[-1].__name__)
    def wrapper(*objects, raise_err=True):
      for o in objects:
        if type(o) in valid_types if reverse else type(o) not in valid_types:
          if raise_err: raise TypeError(message.format(found=type(o).__name__, valid=type_msg))
          return False
      return True
    return wrapper
  return decorator

# Some aliases to avoid overloading of originals
Str = str
Int = int
Float = float
Bool = bool

class Tests:
  @staticmethod
  @_create_test("object must be {valid!r}, not {found!r}", Bool)
  def bool(*objects, raise_err=True): ...

  @staticmethod
  @_create_test("can't convert {found} to {valid}", Int)
  def int(*objects, raise_err=True): ...

  @staticmethod
  @_create_test("can't convert {found} to {valid}", Float, Int)
  def float(*objects, raise_err=True): ...

  @staticmethod
  @_create_test("can't convert {found!r} object to {valid} implicitly", Str)
  def str(*objects, raise_err=True): ...

  @staticmethod
  @_create_test("buffer must be {valid} object, got {found!r} object", Str)
  def buff(*objects, raise_err=True): ...

  @staticmethod
  @_create_test("object {found!r} isn't a {valid}", tuple, list)
  def list(*objects, raise_err=True): ...

  @staticmethod
  @_create_test("{valid} object not allowed", type(None), reverse=True)
  def none(*objects, raise_err=True): ...


  @staticmethod
  def object_size(size, *objects, raise_err=True):
    for o in objects:
      s = len(o)
      if s != size:
        if raise_err: raise IndexError(f"size of {type(o).__name__!r} must be {size}, not {s}")
        return False
    return True

class Convert:
  @staticmethod
  def to_bool(value):
    if type(value) == str: return value.lower() in ("yes","on","enabled","activated","true","1")
```

```
    else: return bool(value)
```

robot\util\vars.py

```
"""Global variables"""
import os, threading


AXIS_JOINTS_COUNT = 6
AXIS_JOINTS_PRECISION = 5

JOINTS_DEFAULT_POSE = [0,0.5,-1.25,0,0,0]
JOINTS_MOVE_OFFSET = 0.1

TRAJECTORY_FILE_EXT = ".trj"
TRAJECTORY_DIR_PATH = "./saved_trajectories"

TRAJECTORY_BANNED_DUPLICATED_STEPS = []
TRAJECTORY_NEEDED_STEPS = []
TRAJECTORY_PROPERTY_STEPS = []

TRAJECTORY_MAX_ID = 2**16-1
TRAJECTORY_IDS = []

EXIST_REQUESTED_EVENT = threading.Event()

MODBUS_SERVER_ADDRESS = "0.0.0.0"
MODBUS_SERVER_PORT = 5022
MODBUS_IDENTITY_VENDOR_NAME = "lycee-ledantec"
MODBUS_IDENTITY_VENDOR_URL = "https://lycee-ledantec.fr/"
MODBUS_IDENTITY_PRODUCT_NAME = "Projet Robotique"
MODBUS_IDENTITY_MODEL_NAME = "Niryo Robot - Modbus Server"
MODBUS_IDENTITY_REVISION = "1.0"

ROSPY_NODE_WAIT_TIMEOUT = 15 # in seconds

LOCK_FILE_PATH = "/tmp/.niryo-project.lock"


def format_trajectory_file_name(name: str) -> str:
  return name + "{sep}{index}" + (TRAJECTORY_FILE_EXT if name.endswith(TRAJECTORY_FILE_EXT) else "")

def get_trajectory_file(name: str) -> str:
  if not name: return None
  if not os.path.exists(TRAJECTORY_DIR_PATH): os.mkdir(TRAJECTORY_DIR_PATH)
  return os.path.join(TRAJECTORY_DIR_PATH, name if name.endswith(TRAJECTORY_FILE_EXT) else name +
TRAJECTORY_FILE_EXT)

def list_trajectory_files(custom_dir: str="") -> 'list[str]':
  dir = custom_dir or TRAJECTORY_DIR_PATH
  if not os.path.exists(dir): os.mkdir(dir)
  return [os.path.join(dir, f) for f in os.listdir(dir) if f.endswith(TRAJECTORY_FILE_EXT)]

```