Université de Genève

Cybernétique appliquée

Robot NAO : Parcours VITA

A. Bulla, bulla3@etu.unige.ch
B. Barbieri, barbieb0@etu.unige.ch
A. Béguin, beguiaa0@etu.unige.ch

Genève, le 17 juin 2011

Ce travail présente en détail le compte rendu du projet de cybernétique 2011. Le concept visait à simuler un parcours VITA à l'aide d'un robot androïde Nao. Ce projet s'articule en deux phases. Premièrement, le robot doit être capable de se déplacer dans un environnement de façon autonome. Cet environnement est construit et contrôlé à l'avance. La connaissance à priori de l'espace de travail facilite l'orientation du robot, puisque les objets qu'il rencontre lui sont connus. En particulier, le robot devra trouver des totems représentant l'emplacement des exercices, et les identifier. Il ne rencontre pas d'obstacles et son champ de vision est relativement neutre. Dans un second temps, il devra effectuer les exercices physiques ou d'équilibre reconnus.

Pour réaliser ces exercices imposés, il convient de les définir préalablement. Cette notion est traitée en détail au point 2. Le déplacement du robot entre les postes est abordé au point 3. Le robot utilise son système de vision embarqué pour repérer les éléments intéressants du décor. L'extraction des caractéristiques (de forme et de couleur) est illustrée à la partie 4. L'implémentation du programme à été réalisée majoritairement en Python, avec une aide sporadique du logiciel *Choregraphe* fourni par *Aldebaran*.

1 Fonctionnalités du robot NAO

Cette partie introduit la dynamique générale du robot NAO et présente brièvement ses capacités. Le caractère volontairement introductif de la section offre au *NAO-néophyte* un aperçu des fonctionnalités utiles à ce projet. Le lecteur aguerri trouvera aisément son chemin à travers ces quelques pages. Il lui est possible de débuter ce rapport à la partie 2 détaillant les mouvements programmés.

Les points suivants abordent plus particulièrement les propriétés des liens du robot, décrivent indirectement l'ensemble des mouvements possibles, et présentent les caractéristiques du système de vision embarqué du NAO. Pour tout complément d'information, il est possible de consulter la documentation officielle [1].

1.1 Squelette et liens principaux

Les figures 1 et 2 montrent la longueur des liens du robot. On peut également y déduire les dimensions générales du squelette.

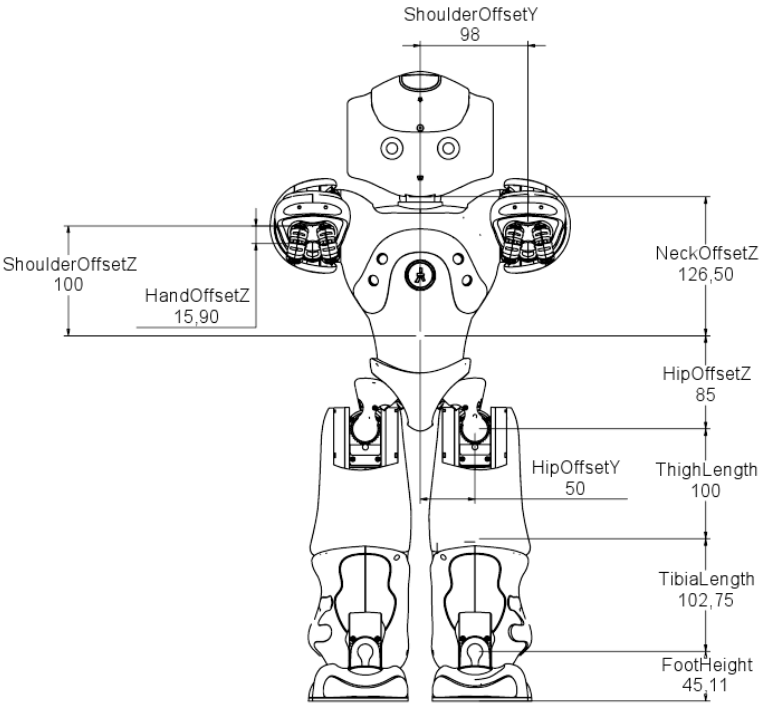

FIGURE 1: Robot vu de face, les bras tendus. Cette position correspond également à la position *zéro* du NAO. Les dimensions, indiquées en millimètres, permettent de déduire la longueur des liens et la position des différents joints.

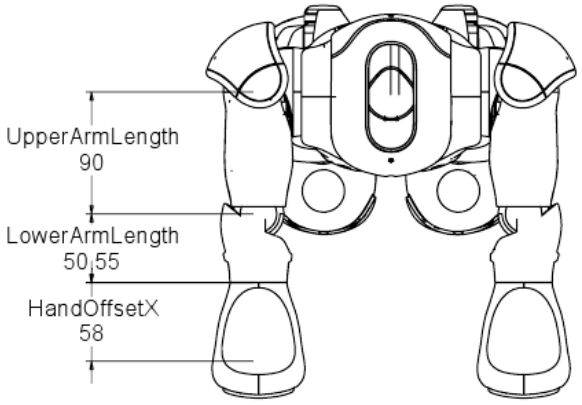

FIGURE 2: Robot vu de dessus. Ce plan offre un détail sur la longueur des membres supérieurs. Les dimensions sont données en millimètres.

La figure 3 indique la position des joints du système. Des moteurs, situés à l'intérieur du corps, actionnent ces articulations à la demande. Il est nécessaire d'asservir préalablement tous les moteurs du robot, avant que l'ordre reçu ne soit exécuté.

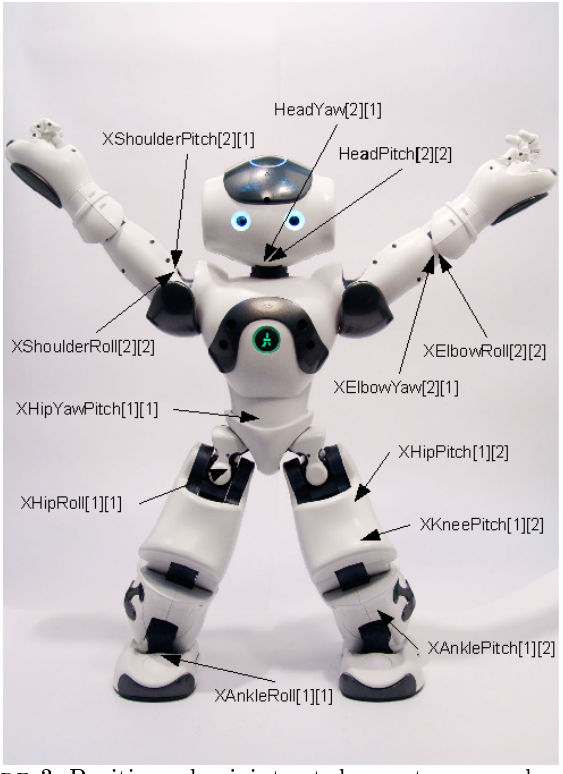

FIGURE 3: Positions des joints et des moteurs sur le robot.

En mode asservi, le NAO peut rester debout et garder l'équilibre. En contrepartie, cela nécessite de dépenser de l'énergie, ce qui diminue inévitablement la durée de vie de la batterie. Si l'on désire utiliser le NAO sans le déplacer, il est parfois judicieux de l'installer dans une configuration stable, par exemple en position assise. Désasservir ses moteurs permet alors d'économiser de l'énergie et prolonger la durée de vie des joints.

En effet, ceux-ci ont une fâcheuse tendance à chauffer (fait probablement lié au fonctionnement des moteurs). A force, ce comportement est néfaste pour les joints, et le robot risque d'émettre un avertissement d'arrêt d'urgence. Dans les cas plus extrêmes, le robot peut se mettre hors tension. Pendant le développement du projet, la situation précédente s'est produite à différentes reprises. En particulier, si les joins sont trop souvent sollicités pendant les phases d'exercices physiques intenses, leur température s'élève dangereusement.

La problématique précédente à donné naissance à une idée intéressante. Étant donné que des contraintes matérielles et énergétiques nous forcent à effectuer de temps à autre une pause, pourquoi ne pas intégrer ces événements dans le déroulement naturel du programme. Par analogie avec un véritable parcours sportif, le robot pourrait réagir au message de fatigue (nécessité de se reposer ou de se recharger), en adoptant une position où les moteurs pourront être désasservis sans crainte. Au sein de notre programme, on simule un tel état à partir d'un certain nombre d'exercices effectués (voir point 3).

Bien évidemment, dans un cas réel, l'état des joints et de la batterie peut être récupéré à travers un module interne. Une fois ces messages interprétés, la posture assise est adoptée jusqu'à ce que la température des joints soit redescendue. Dans le cas réel de gestion d'énergie, la solution automatique est moins satisfaisante, puisqu'il est notamment impossible de recharger le robot lorsque celui-ci est sous tension (voir point 1.4 pour plus de détails).

1.1.1 Articulations et moteurs

Les articulations et leur sens (mouvements possibles) sont détaillées à la figure 4. Le modèle plus formel de la figure 5 montre en détail la hiérarchie des axes.

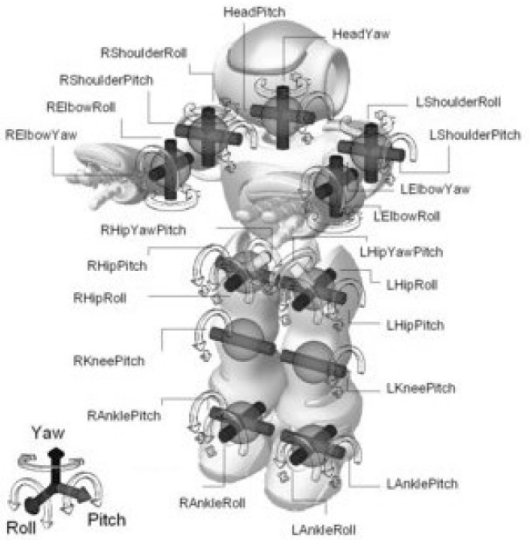

FIGURE 4: Axe et orientation des articulations du robot.

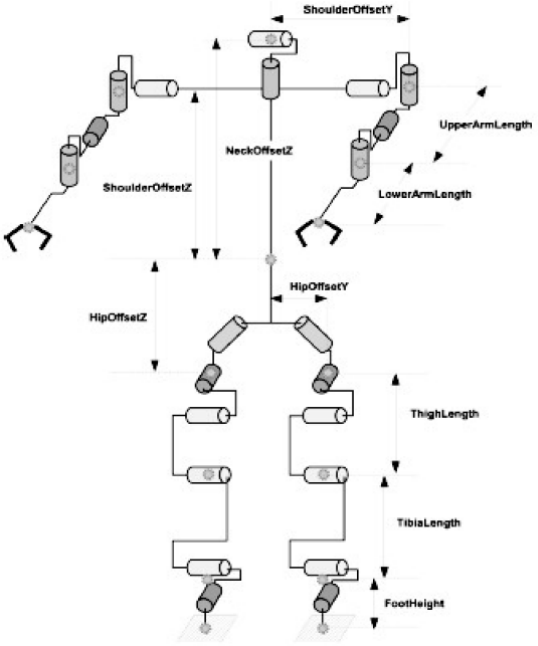

FIGURE 5: Détail de la hiérarchie des axes.

Toutes les articulations peuvent être pilotées de manière autonome. Pour cela, différentes approches sont possibles. Soit, le programme est écrit en Python (ou C++), auquel cas on utilise le module *ALMotion* fourni avec la SDK d'*Aldebaran*. L'autre option est de réaliser des mouvements complexes avec le logiciel *Choregraphe* et de les exécuter.

Pour ce projet, après avoir tenté une approche essentiellement basée sur *Choregraphe*, nous avons opté pour la version Python. Ce choix nous a permis d'augmenter la flexibilité du programme, et d'obtenir une plus grande liberté au niveau du code. Une meilleure séparation (modularité) entre les programmeurs à également fait pencher la balance en faveur de cette méthode de développement.

1.1.2 Logiciel *Choregraphe*

Il est également possible d'adopter une approche hybride, c'est-à-dire de créer un mouvement à l'aide de *Choregraphe*, puis d'importer les coordonnées des joints directement dans le programme Python. On fixe alors la position des joints à chaque *frame* de la *timeline*. Le lissage des trajectoires entre les différentes *frames* se fait par interpolation (bézier par exemple). La figure 6 est une capture d'écran de l'interface principale pour la définition des mouvements. En utilisant le module *ALBehviorManager*, il est possible d'importer un projet *Choregraphe* complet en Python (ou C++). Nous avons réalisé cela pour exploiter les box "SitDown" et "StandUp" et faire asseoir et lever le robot en fonction de sa position actuelle. Les figures suivantes montrent l'interface de commande pour certains des joints.

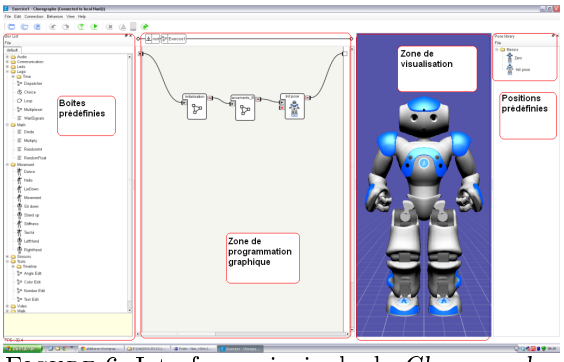

FIGURE 6: Interface principale de *Choregraphe*.

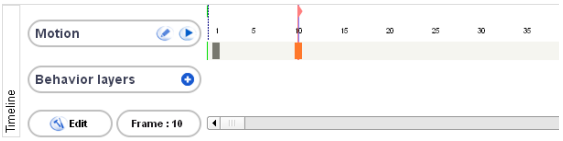

FIGURE 7: Timeline pour la définition des mouvements.

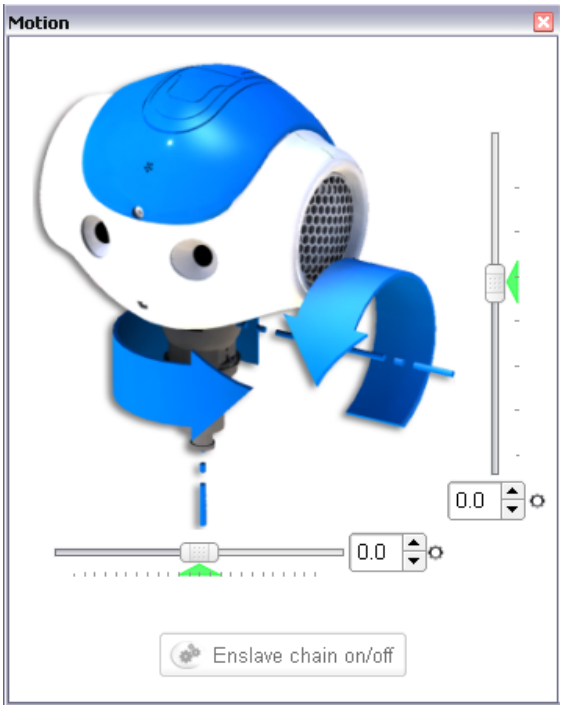

FIGURE 8: Mouvements des joints de la tête.

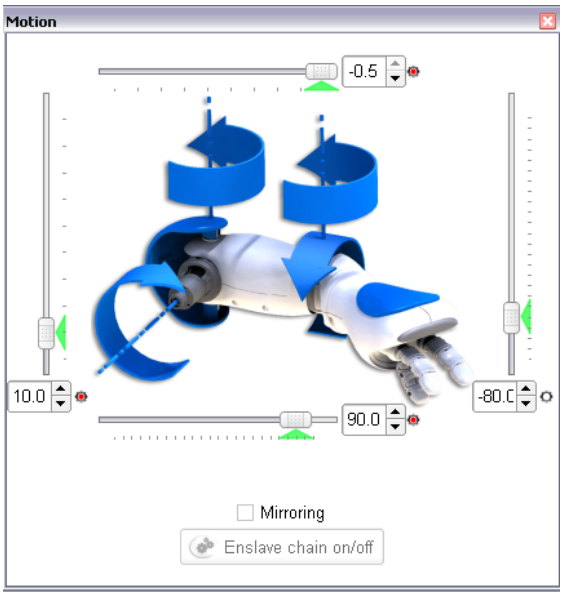

FIGURE 9: Mouvements des joints du membre supérieur.

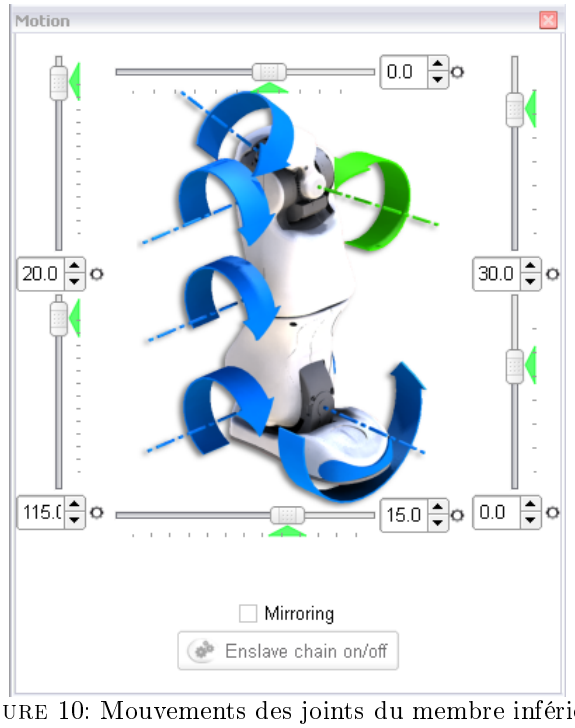

FIGURE 10: Mouvements des joints du membre inférieur.

1.2 Amplitudes de mouvement des membres

Les figures ci-dessous indiquent les angles limite des articulations du robot.

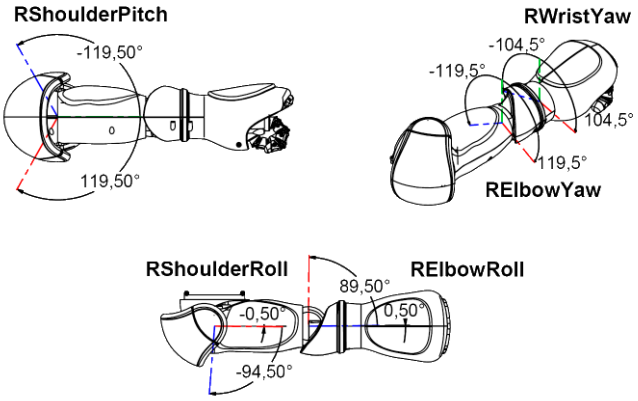

FIGURE 11: Amplitude des bras.

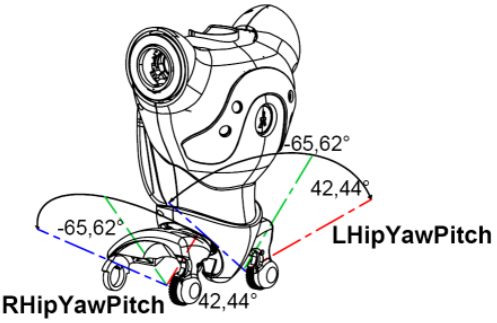

FIGURE 12: Amplitude des mouvements au niveau des hanches.

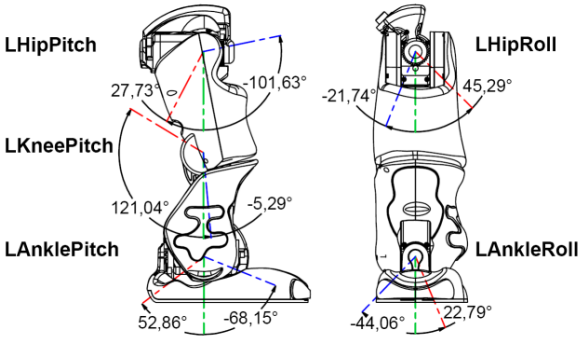

FIGURE 13: Amplitudes pour les pieds.

1.2.1 Vision et tête du robot

La plupart de la connectique du robot se situe au niveau de sa tête. La vision du NAO est composée de deux caméras, l'une positionnée au niveau du front, l'autre dans sa bouche. Il est possible de sélectionner l'une des deux caméras pour prendre une image et l'envoyer depuis le robot (ou encore de la traiter en local). Pour réaliser ces interactions, on utilise le module *ALVideoDevice* de la SDK. A noter que pour ce projet, il nous a été impossible d'obtenir un flux vidéo depuis la caméra (bien qu'en théorie cela devrait être possible). Nous avons donc traité les informations en prenant successivement plusieurs images aux moments clé du parcours.

La figure 14 montre l'angle de vue des deux caméras. Pour ce projet, dans un premier temps, nous utilisions la caméra supérieure pour détecter les objets à longue distance, et la caméra inférieure pour trouver les objets aux pieds du robot. Ceci dit, l'angle de vue de la caméra basse est parfois trop limité. Dans certains cas, comme aux abords des totems (marqueurs pour les postes du parcours), nous avions besoin d'une portée intermédiaire. Afin de traiter ce cas particulier : nous avons donc finalement adopté la combinaison de mouvements suivante : baisser la tête, prendre une image avec la caméra supérieure, rechercher les objets bleus (à moyenne portée), relever la tête, prendre une nouvelle image avec la même caméra, puis détecter les objets totems à rouges (longue portée).

La figure 15 indique les angles limites que nous devons respecter lorsqu'il est nécessaire d'incliner la tête du robot (*Head Pitch*).

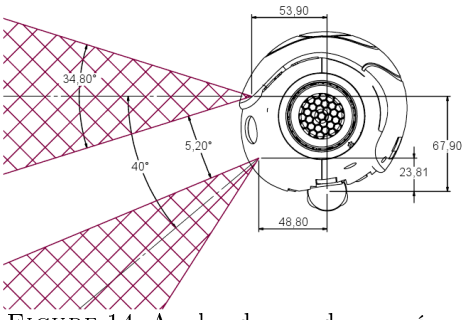

FIGURE 14: Angles de vue des caméras.

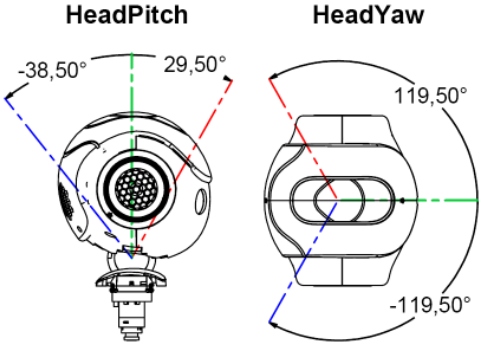

FIGURE 15: Inclinaison et rotation de la tête.

Outre le système de vision, le robot NAO est également capable de reproduire des sons (.wave ou .mp3), ou encore de synthétiser des paroles à l'aide de son module *Text-to-Speech* (TTS). Le son est émis au niveau de ses "oreilles" grâce à des hauts-parleurs.

Nous avons utilisé cette fonctionnalité pour faire parler le NAO lorsqu'il se retrouve dans une posture particulière. Par exemple, face à un totem, la phrase "I found the red totem!" nous indique que le robot a identifié sa cible, et va avancer en sa direction.

Le module responsable de la synthèse TTS est *ALTextToSpeech*. Il faut cependant parfois utiliser une phrase écrite phonétiquement, pour obtenir l'effet souhaité. Pour reproduire la phrase de l'exemple précédent, la chaîne de caractères envoyée au module est en réalité : "I found the raid to-taim!".

1.3 Connexion réseau

La connexion avec l'extérieur peut être réalisée à travers une interface sans fil (*wifi*), via *bluetooth* ou encore à l'aide d'une connectique filaire (située derrière la tête du NAO). Nous avons employé la solution *wifi* pour communiquer avec le robot, lui envoyer des ordres et récupérer ses photos.

Le programme est ainsi exécuté sur une machine distante à l'aide d'un interprète Python. A priori, passer d'une exécution distante à une exécution embarquée ne devrait pas nous forcer à modifier une grande partie du code. Les services du robot doivent être accédés via un proxy. Ce choix de design garantirait ainsi un comportement quasi-identique où que le programme soit exécuté.

Cependant, force est de constater qu'il ne suffit pas de seulement modifier l'adresse IP du proxy, comme on pourrait s'attendre. Le module d'imagerie (*ALVideoDevice*) par exemple n'utilise pas la même méthode pour capture une image en local ou à distance. Ceci est un peu dommage, car cela indique que les choix de design du software ne sont pas complètement aboutis et/ou assumés dans l'état actuel des choses.

Un autre problème de nature matérielle est survenu à plusieurs reprises lors du développement. Le robot ne réussit parfois pas à trouver le point d'accès sans fil (routeur *wifi*). Le seul moyen est alors de s'interfacer avec lui par cable RJ45, et de le forcer à accepter la connexion *wifi* manuellement.

1.4 Batterie

Un problème particulièrement important est la durée de vie de la batterie. En effet, selon les mouvement effectués et leur cadence, celle-ci s'use très rapidement. Attention donc à ne pas laisser sous tension le robot pendant une durée trop importante. Il est également recommandé d'avoi toujours une batterie de rechange à proximité (chargée).

2 Exercices du parcours VITA

Dans cette partie du projet, il convient tout d'abord de définir les exercices de parcours Vita que le robot NAO devra reproduire. Pour ce faire, il faut tenir compte des capacités du dit robot. Une fois les exercices choisis, ceux-ci sont créés et testés une première fois à l'aide d'une interface graphique nommée *Choregraphe*. Vient ensuite les tests en grandeur réelle.

2.1 Choix des exercices

L'idée première est de se baser sur des exercices réels. Ainsi, les panneaux officiels du parcours VITA de Zurich ont été sélectionnés. Cependant, le choix des exercices a été plus compliqué que prévu. En effet, beaucoup d'exercices requièrent soit des mouvements irréalisables par le robot Nao soit une force qu'il n'a pas. Il y a par exemple, les tractions, les mouvements de hanches, de chevilles, les pompes, les sauts, la course à pieds, les exercices avec des anneaux ou des barres parallèles et sûrement d'autres encore. Une fois ce tri fait, il ne reste malheureusement plus beaucoup de diversité au niveau des exercices. Il a donc été choisis de se rabattre sur les exercices d'étirements, ainsi six exercices ont été retenus :

1. Figure 16 : Exercice travaillant la mobilité des bras. Le robot Nao doit effectuer des mouvements de balancier d'avant en arrière avec ses bras.
2. Figure 17 : Deuxième exercice de mobilité des bras. Les bras doivent former une ligne horizontale avec les épaules, puis doivent être levés tout en restant tendu de manière à ce que les mains se retrouve au dessus de la tête.
3. Figure 18 : Exercice d'étirements des jambes. Le robot Nao doit se tenir en équilibre sur une jambe et plier la deuxième en ramenant la main correspondante proche de la cheville de la jambe pliée pour simuler un étirement de la cuisse.
4. Figure 19 : Exercice d'étirement de la nuque. Le robot Nao mets les bras derrière le dos et effectue des mouvements de tête de gauche à droite.
5. Figure 19 : Deuxième exercice d'étirement de la nuque. Cette fois les mouvements de la tête se font de haut en bas.
6. Figure 20 : Cet exercice simule un étirement de l'intérieur des cuisses. Une jambe tendue en arrière tandis que la deuxième est légèrement plié vers l'avant. Le robot Nao doit plier encore plus la jambe avancée tout en gardant l'équilibre.

Il est à noter que les exercices d'étirements des jambes ont été implémentés pour les deux côtés.

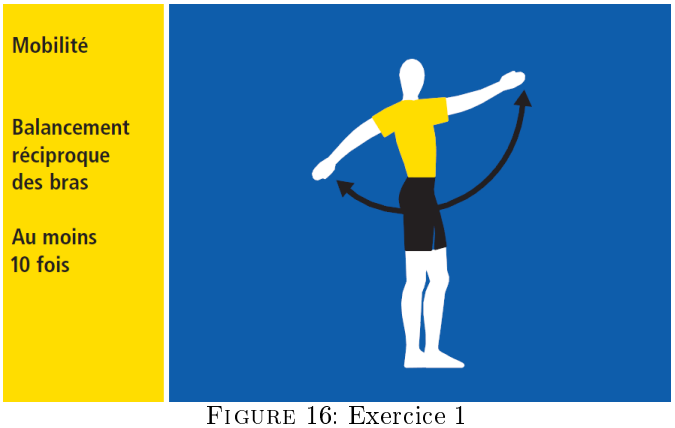

FIGURE 16: Exercice 1

FIGURE 17: Exercice 2

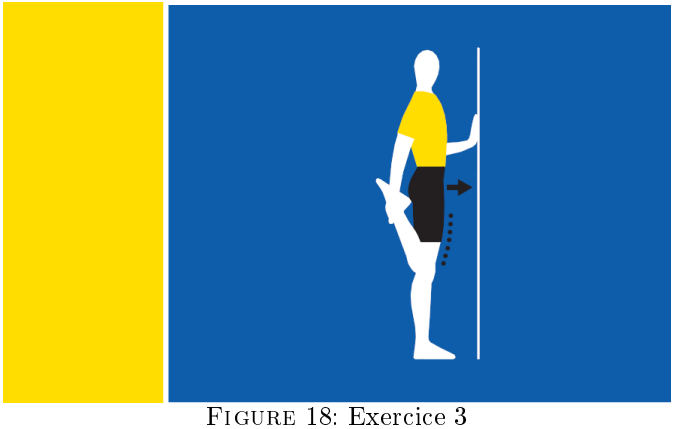

FIGURE 18: Exercice 3

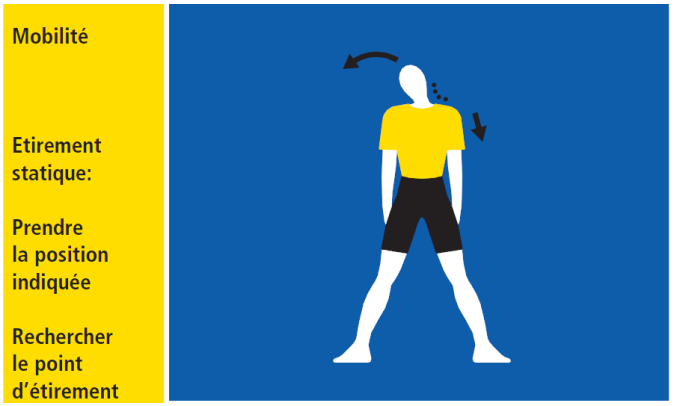

FIGURE 19: Exercice 4 et 5

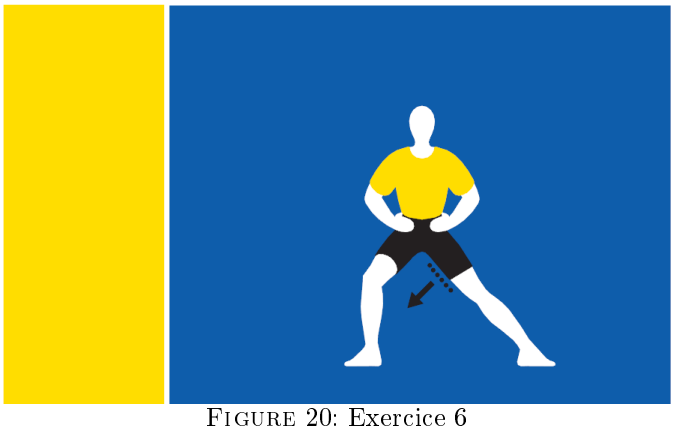

FIGURE 20: Exercice 6

2.2 Définition d'un mouvements

Il est possible de définir un exercice, ou plus généralement un mouvement, de trois manière différentes :

1. Faire bouger manuellement le robot en enregistrant les mouvements dans une ligne temporelle (*timeline*) du logiciel Choregraphe
2. Définir des *keyframes* dans une *timeline* du logiciel Choregraphe auxquelles sont associés des positions des différentes articulations définies grâce à l'interface graphique fournie par le logiciel.
3. Calculer et affecter manuellement avec un langage de programmation textuel les valeurs des angles de chaque articulations à des temps donnés.

Voyons plus en détails ces trois méthodes.
La première est très imprécise car il est nécessaire de désasservir les moteurs afin de pouvoir bouger manuellement les membres du robot. Cependant il est très difficile de faire garder l'équilibre au robot, ou le buste droit au robot tout en essayant de lui faire bouger d'autres parties du corps. Les mouvements ainsi définit sont imprécis, rarement fluides et prennent beaucoup de temps pour un résultat moyen.

La deuxième méthode est la moins fastidieuse à utiliser. Elle permet de définir les positions (angles) des articulations de manières plus intuitive et avec un retour visuel immédiat dans Choregraphe. Ainsi, il est possible de se rendre rapidement compte d'éventuelles erreurs et ce sans risquer d'endommager le robot, contrairement aux tests réels. Les plus grandes difficultés avec cette méthode sont la gestion de l'équilibre, puisqu'il n'y a pas vraiment d'indication concernant ce paramètre dans le logiciel. La deuxième difficulté vient de l'absence de sol "statique" dans la fenêtre de simulation. Il est sous-entendu par là que pour un mouvement où le robot doit plier les jambes, dans l'environnement graphique de Choregraphe, le robot ne donnera pas l'impression de se baisser (le centre de masse se rapproche du sol) mais plutôt celle de lever les jambes (le centre de masse ne bouge pas, les pieds ne sont plus en contact avec le sol virtuel).

La troisième méthode proposée est beaucoup plus précise, mais requiert un temps de mise en place gigantesque. Comme on peut le voir au point **??**, le robot possède un grand nombre d'articulations. Les calculs des angles devant être modifiés pour composer un mouvement se compliquent très rapidement. De plus, il est difficile de se représenter mentalement un mouvement par une suite d'angle et d'articulations.

Après avoir essayé ces trois méthodes, il a été choisis d'utiliser Choregraphe et sa *timeline* pour définir les exercices, puis de les exporter en Python sous la forme de vecteurs. Pour définir un mouvement, trois vecteurs sont nécessaires :
- un vecteur de nom contenant les noms des articulations utilisées dans le mouvement

- un vecteur de positions spécifiant les positions des articulations du premier vecteur
- un vecteur de temps contenant les temps auxquels les articulations du premier vecteur doivent être aux positions du second vecteurs

Une fois toutes les valeurs définies, une méthode d'interpolation du proxy *ALMotion* est appelée.

En python cela donne :

Définition des valeurs de l'articulation HeadYaw et appel de la méthode d'interpolation

```
names = list()
times = list()
keys = list()

names.append("HeadYaw")
times.append([ 0.10000])
keys.append([ [ 0.00000, [ 2, -0.03333, 0.00000], [ 2, 0.00000, 0.00000]]])

try:
   self.__proxy.angleInterpolationBezier(names, times, keys);
except BaseException, err:
   print str(err)
```

2.3 Connexion au robot Nao

Un objet Python a été implémenté pour gérer la connexion au robot Nao ainsi que les différents proxys utiles pour le commander. Tous ne sont pas utilisés dans ce projet, mais la une grande partie des proxys sont accessibles grâce à la méthode `Connexion.getProxy(nom_proxy)` qui créé et retourne le proxy correspondant au nom passé en paramètre. Les noms des paramètres sont les mêmes que les noms des modules du Nao donnés dans la documentation.

Connexion.py

```
from naoqi import ALProxy

class Connexion(object):
    '''
    @contructor
    '''
    def __init__(self, robot):

        # choix du robot
        if robot == None:
            self.__ip = "laptopnono.local" # en local
        else:
            if robot == "astro":
                self.__ip = "10.194.70.11" # astro
            else:
                self.__ip = "10.194.70.12" # robby

        # port par defaut
        self.__port        = 9559

        # declaration des proxy
        self.__AudioDeviceProxy     = None
        self.__AudioPlayerProxy     = None
        self.__BehaviorManagerProxy = None
        self.__FrameManagerProxy    = None
        self.__FsrProxy             = None
        self.__InertialProxy        = None
        self.__LedsProxy            = None
        self.__LoggerProxy          = None
        self.__LogoDetectionProxy   = None
        self.__MemoryProxy          = None
```

```
        self.__MotionProxy          = None
        self.__SensorsProxy         = None
        self.__SonarProxy           = None
        self.__TextToSpeechProxy    = None
        self.__VideoDeviceProxy     = None
        self.__DCMProxy             = None


    '''
    @group: Setters
    '''
    def setIP(self, ip):
        self.__ip = ip

    def setPort(self, port):
        self.__port = port

    '''
    @group: Getters
    '''
    def getIP(self):
        return self.__ip

    def getPort(self):
        return self.__port

    def getProxy(self, name):
```

2.4 Les exercices

Les mouvement abordés précédemment ne sont qu'une partie d'un exercice. En effet, un mouvement est toujours définit par rapport à une position initiale. En fait de manière plus générale encore, un mouvement entre deux positions (états) est une interpolation des positions en fonctions du temps entre la position de départ et de celle d'arrivée. Il est donc nécessaire de s'assurer que le robot Nao sera dans la position adéquate avant de commencer l'exercice et en le terminant afin qu'il puisse ensuite se déplacer correctement ou faire un autre exercice. Un exercice est donc décomposé en trois parties :

1. initialisation : positionne le robot de manière à pouvoir commencer un exercice
2. action (do) : le robot effectue l'exercice un nombre de fois donné
3. finalisation : remet le robot dans une position telle qu'il puisse reprendre ses occupations

Ainsi, chaque exercice est représenté par un objet dont la méthode principale est *do(nbFois)* :

Ex1.py

```
# ordonne au nao de faire l'exercice nbFois fois
def do(self, nbFois):

   if nbFois > 0:

      # initialisation
      self.initialisation()

      # exercice
      self.repeatExercice(nbFois)

      # terminaison
      self.finalise()
```

Le proxy est passé en paramètre à l'initialisation de l'objet :

```
def __init__(self, p):
    '''
    Constructor
    '''
    self.__proxy = p
```

De manière plus précise, le robot Nao n'a pas connaissance de sa physionomie. Ainsi, il peut exister des positions de départ et d'arrivé correctes, cependant en utilisant l'interpolation simple fournie sur le robot Nao, ces membres peuvent entrer en collisions, voir se bloquer. Il est donc nécessaire pour beaucoup de mouvements de les découper en une suite de mouvements. Ci-dessous, un exemple illustrés du découpage d'un mouvement afin de faire conserver son équilibre au robot et d'éviter toute collision. Tout d'abord la *timeline* avec les différentes *keyframes* représentants les positions composants le mouvement.

FIGURE 21: Découpage d'un mouvements

La figure 22 montre les différentes positions par lequel le robot devra passer pour effectuer son mouvement correctement.

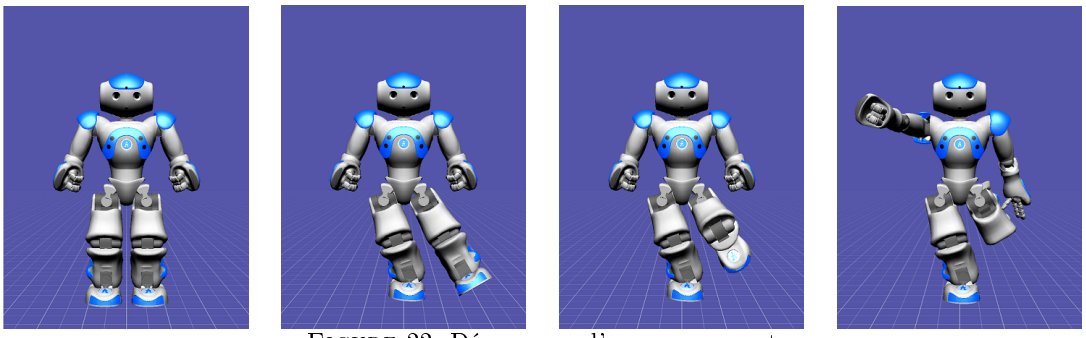

FIGURE 22: Découpage d'un mouvement

3 Parcours

Le but principal de l'application est de proposer un système de guidage automatique pour le robot. Celui-ci doit pouvoir se mouvoir à l'intérieur d'un espace dont les caractéristiques sont contrôlées à l'avance.

L'environnement du robot est relativement neutre. En particulier, on détermine à l'avance les objets rencontrés et leur positions. Les totems rouges (voir figure 23) indiquent les postes vers lesquels le NAO devra se déplacer. Une fois arrivé au pied d'un totem, un ou plusieurs objets bleus lui indiquent l'exercice à effectuer.

Il peut y avoir plusieurs totems rouges. Ceux-ci sont généralement disposés en cercle à environ six pas d'un centre fictif. Ils doivent être séparées par une distance acceptable, sans quoi le robot, placé au centre, pourra confondre ces cibles.

Une fois un totem repéré le NAO tentera d'ajuster sa trajectoire jusqu'à ce qu'il voie le totem de front (voir la partie 4 sur l'imagerie). Puis, il fera un certain nombre de pas selon la distance qui lui reste à parcourir. Cette quantité est estimée par la taille approximative du totem sur la dernière image capturée. Les derniers pas (trois en moyenne) sont parcourus seulement dans le cas où aucun objet bleu n'est repéré à moyenne portée.

Si tel est le cas en revanche, cela veut dire que le robot est tout près du pied d'un totem. A priori ceci constitue une hypothèse raisonnable de travail. Initialement, nous avions utilisé des *NAOMarks* disposées au pied de chaque totem pour indiquer le numéro de l'exercice à effectuer. Cependant, le NAOne réussit pas toujours (presque jamais en réalité) à déterminer la marque correcte. Par conséquent, nous avons remplacé ces motifs par un nombre quelconque de cubes (entre 1 et 6). Cette solution nous a semblé être le meilleur compromis possible, en particulier parce qu'il était ainsi possible de réutiliser une partie de l'imagerie déjà implémentée pour la détection du totem.

Le robot identifie l'exercice à effectuer en comptant le nombre de cubes bleus sur l'image. Il effectue ensuite un demi-tour et invoque l'exercice désiré depuis une collection d'objets (voir partie 2 pour leur définition). Il décrémente sa jauge d'énergie en fonction de l'exercice effectué. Pour terminer, il retourne environ au centre du cercle (effectue six pas) et se retourne face à la scène (angle approximatif de 120 degrés).

Les images des figures 23 et 24 montrent les objets présents dans l'environnement. La figure 25 indique un cheminement de parcours classique entre les différents postes.

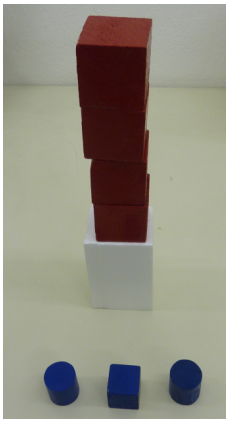

FIGURE 23: Un poste du parcours : le totem rouge et ses cubes bleus. Ceux-ci indiquent dans cet exemple l'exercice no3.

Dans le cas où la jauge d'énergie est entièrement dépensée après un exercice, le robot se repose (le programme s'interrompt) pour un certain temps. Pour cela, nous avons choisi de poser le robot dans une position assise et de désasservir momentanément ses moteurs. La stature assise

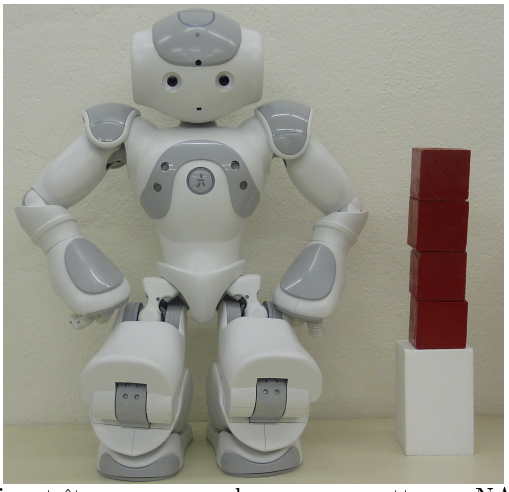

FIGURE 24: Les totems doivent être assez grands pour permettre au NAO de les apercevoir depuis suffisamment loin.

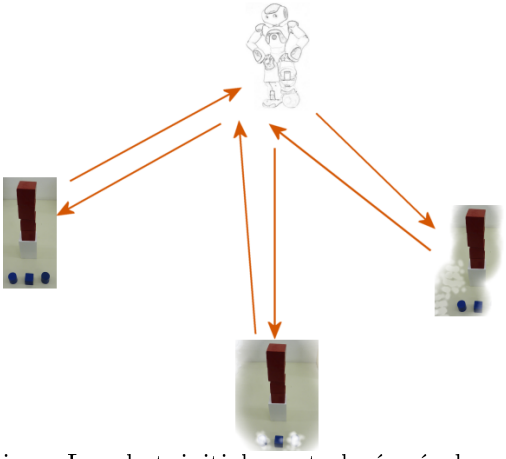

FIGURE 25: Parcours classique. Le robot, initialement placé prés du centre, rejoint chaque totem puis revient à sa position centrale.

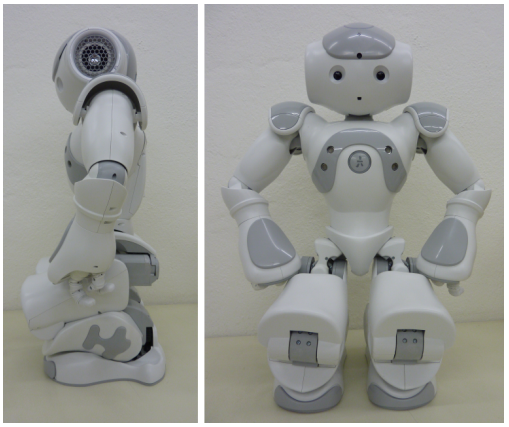

FIGURE 26: Posture stable. Le robot est assis, ses membres supérieurs s'appuyant sur ses genoux pour garder sa stabilité

doit être stable et permettre au robot de tenir sans tomber. Nous avons choisi d'utiliser ses membres supérieurs pour appuyer le haut du corps sur les genoux. Ainsi son buste ne tombe pas vers l'avant lorsque les moteurs sont désasservis.

Le programme s'arrête totalement après avoir effectué un certain nombre d'exercices.

4 Imagerie

Dans notre projet, le robot devra effectuer un parcours vita, ce qui correspond essentiellement à une série d'exercices différents une série de lieux. L'analyse d'image aura donc deux objectifs différents : premièrement, reconnaître dans le champ de vision du robot les lieux des exercices, typiquement dans un parcours vita il s'agira de retrouver un panneau. Ensuite, il faudra différencier ces lieux pour reconnaître l'exercice que le robot devra effectuer. Dans un vrai parcours vita, l'être humain va se rendre au panneau de l'exercice et lire la description de l'exercice sur le panneau. C'est le comportement que nous aimerions reproduire.

Notre idée de départ était de seuiller l'image rendue par la caméra et de détecter des régions, puis éventuellement d'utiliser les propriétés de ces régions, pour trouver les zones carrées d'une couleur (c'est-à-dire, les panneaux des exercices). Nous avons également essayé le module de détection de landmarks inclut avec le Nao, qui peut détecter et diférencier un certain nombre de symboles (landmarks), dans le but de différencier les exercices. En fin de compte, il s'est avéré plus simple de simplement détecter les régions de certaines couleurs pour les deux tâches, comme je l'expliquerai plus loin.

4.1 Détection des panneaux

4.1.1 Seuillage

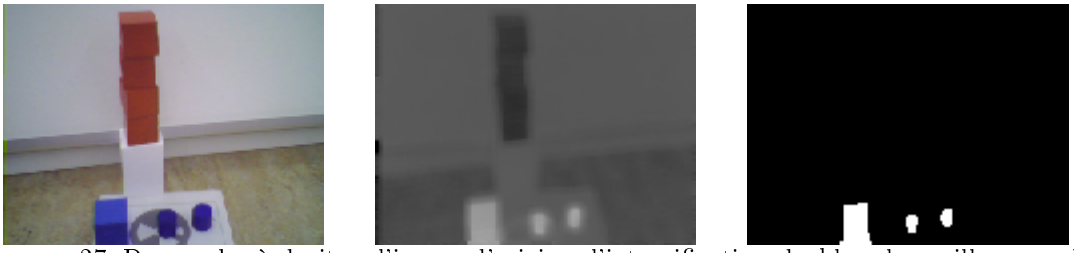

FIGURE 27: De gauche à droite : l'image d'origine, l'intensification du bleu, le seuillage sur le bleu

Nous avons d'abord travaillé avec un seuillage fixe sur le plan bleu ou rouge de l'image, une solution bien évidemment pas idéale puisque très dépendante de la luminosité de la pièce. Nous avons donc amélioré notre méthode en commençant par intensifier la valeur de la couleur recherchée avec la formule, pour le rouge par exemple $\frac{r}{(r+g+b)}$. Ensuite de quoi nous avons seuillé en fonction de l'histogramme de l'image ainsi retournée, en prenant une valeur de seuil à 85% de la valeur maximale de cette image. Nous espérons ainsi que tous les objets, par exemple, d'un rouge pétant, ressortiront sur l'image alors que le fond, le reste de l'image disparaîtra. Une petite modification encore à ajouter à cela est un minimum : si la plus grande valeur de l'image est inférieure à, par exemple, 120 (une valeur que nous avons choisie expérimentalement), on peut conclure que l'image ne contient aucune région, et retourner une image entièrement noire.

4.1.2 Détection de régions

Sur notre image seuillée, donc binaire, on va vouloir détecter les régions. On peut commencer par une simple erosion pour diminuer le bruit qui nous aurait échappé lors du seuillage. Ensuite, pour détecter les régions, nous avons simplement effectué un "region growing", c'est à dire pour chaque pixel blanc cherché tous les pixels blancs voisins, récursivement, jusqu'à avoir trouvé l'ensemble des points connexes. On obtient ainsi les différentes régions de la couleur voulue. Une fois cela fait, on peut déjà trier les régions, encore dans une tentative de réduire le bruit, en retirant les régions trop petites par rapport à la taille de l'image.

4.1.3 Mesures sur les régions

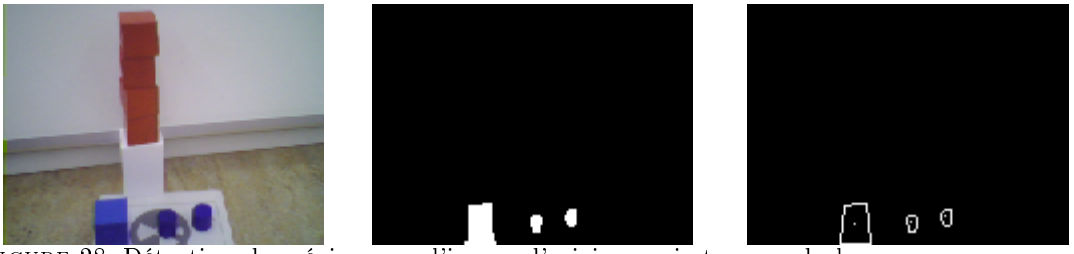

FIGURE 28: Détection des régions sur l'image d'origine, puis traçage de leur coque convexe et centre de masse

Pour pouvoir utiliser les régions détectées, on va vouloir calculer certaines informations sur ces régions. Par exemple, pour pouvoir lui donner une position ou encore effectuer d'autres calculs, on pourrait vouloir calculer son centre de masse. Pour cela, j'ai simplement fait la moyenne des coordonnées de tous les points dans la région. Ensuite pour obtenir la coque convexe, on utilise l'algorithme du "Gift wrapping", aussi connu comme la marche de Jarvis, c'est-à-dire que pour chaque point, on choisit le point suivant de telle sorte que tous les autres points soient du même côté de la droite passant entre ces deux points, jusqu'à avoir fait le tour de la région.

Le but au départ était de trouver un carré bleu représentant un panneau, dans cette optique j'ai également mesuré ce que j'ai appelé la taille de la "plus grande diagonale", c'est à dire la droite qui relie les deux points les plus éloignés de la coque convexe. J'avais dans l'idée que pour un carré, le carré de la diagonale est égal au double de l'aire (ou peut se le représenter assez facilement géométriquement), et donc j'ai comparé le carré de cette diagonale à la taille de la région (le nombre de points), avec une certaine marge, pour détecter les carrés. Nous verrons plus tard que cela ne s'est pas avéré nécessaire.

4.1.4 Remarque sur la détection des carrés

Dans notre implémentation finale, nous avons remplacé les grands panneaux carrés par des sortes de "totems" rouges. Comme je l'expliquerai plus tard, nous avions essayé le module de détection de landmarks du Nao, et pensions simplement placer un landmark au sol devant le totem, plutôt que d'implémenter nous-même une classification des panneaux. Nous n'avons donc pas besoin d'un panneau large pour pouvoir afficher l'exercice dessus. En plus, en utilisant les cubes rouges à notre disposition, nous arrivions finalement assez facilement à différencier les totems du reste de l'image. Nous avons donc opté pour ces structures plus étroites, ce qui nous donnait également une meilleure précision quand à leur position verticale : sur une région ou tous les points sont dans un rectangle fin, le centre de masse sera contenu dans ce rectangle, alors que sur un carré, si par malchance, une certaine luminosité, un reflet ou du bruit quelconque nous faisait manquer une partie de la région, il y'aurait peut-être plus de chance que le centre de masse soit décalé sur la droite ou la gauche du carré. Nous avons donc uniquement utilisé la détection de région, qui s'est avéré tout à fait satisfaisante pour notre application.

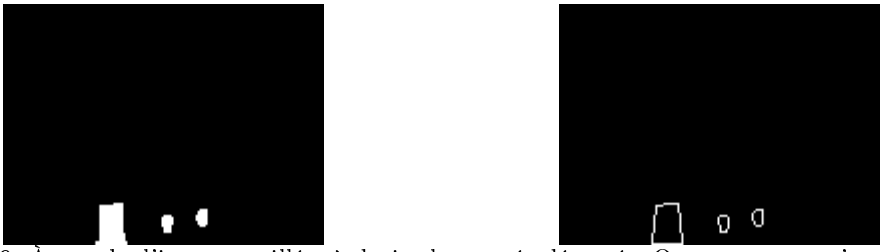

FIGURE 29: À gauche l'image seuillée, à droite les carrés détectés. On remarque qu'avec la faible résolution de notre application, détecter des carrés devient plus difficile.

4.1.5 Application des mesures

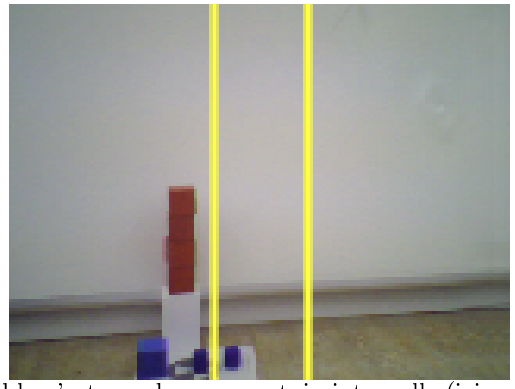

FIGURE 30: Tant que la cible n'est pas dans un certain intervalle (ici en jaune) le robot va tourner dans sa direction

Dans notre application finale, nous demandons au robot de chercher le totem rouge devant lui et d'aller vers lui. Pour cela, nous utilisons la détection de régions rouges, et nous nous basons sur le centre de masse de la région détectée pour aligner le robot horizontalement. Il va donc continuer à tourner en direction du centre de masse trouvé jusqu'à ce que l'image qu'il prenne soit dans un certain intervalle de distance par rapport au centre de l'image prise.

FIGURE 31: Un totem très proche. On remarque que le bas du totem est coupé.

Nous avons également utilisé la taille de la région rouge détectée (c'est-à-dire son aire) pour estimer la distance du robot au totem. En effet, nous voulions que le robot avance d'une distance plus petite entre chaque photo à mesure qu'il s'approche de l'objectif. Nous avons donc fait quelques mesures sur des images prises à des distances typiques, afin de pouvoir choisir la taille du déplacement en fonction de l'image prise.

Lors de la présentation, cette dernière fonction n'a pas très bien marché : le robot avançait trop alors qu'il était très proche. Il est possible que cela soit du à la proximité avec le totem : si l'image ne prend pas la totalité du totem, il se peut que l'aire résultante soit plus petite, et donc que l'estimation de la distance à parcourir soit faussée.

4.2 Différenciation des exercices

4.2.1 Remarque sur la détection des *landmarks* du Nao

Lorsque nous avons découvert cette fonctionnalité du Nao, nous avons tout de suite pensé l'utiliser pour différencier nos exercices. Nous l'avons d'abord essayé séparément, et elle semblait convenir à nos besoins : le module de détection était capable de nous retourner le numéro identifiant du landmark qu'on plaçait devant sa caméra. Nous avons donc essayé cette méthode.

Nous voulions que le Nao détecte les landmarks pendant sa marche : l'idée était que le Nao chercherait devant lui le totem rouge, puis avancerait jusqu'à ce qu'il voie au sol un landmark. Pour faciliter le processus, nous capturons un certain nombre de fois les landmarks devant le

FIGURE 32: Voici un landmark du Nao. Dans cette faible résolution, ils sont peut-être plus difficiles à différencier

Nao, puis prenons le numéro qui a été trouvé le plus de fois, avec un certain seul minimum ; on évite ainsi que le Nao retourne un landmark faux lorsqu'il a simplement trouvé du bruit et s'est trompé. En pratique, nous nous sommes confrontés à plusieurs problèmes. Premièrement, le robot trouvait assez rapidement des landmarks devant lui alors qu'il n'y en avait pas. Nous avons supposé que cela pouvait être du au bruit et à la résolution très faible de l'image : nous avions utilisé la plus petite résolution de la caméra, pour éviter le temps d'échange entre la caméra et le PC relativement long. Il est aussi possible que le fait de prendre les images alors qu'il est en mouvement réduisse la qualité du résultat. Nous avons donc essayé plusieurs autres méthodes, comme par exemple arrêter le Nao pour lui faire reprendre une image en haute résolution chaque fois qu'il pense avoir trouvé quelque chose. Même alors, le robot aura quand même trouvé dans certains cas des landmarks là ou il n'y en avait pas, mais le problème suivant était que le robot n'identifiait pas ou mal la landmark au bout du parcours. Plusieurs fois, il continuait sa course, ne trouvant pas de landmark, même en l'obligeant à s'arrêter et en chercher une à des intervalles réguliers ; d'autres fois, il ne trouvait simplement pas la bonne landmark, et retournait un numéro erronné.

4.2.2 Détection d'objets bleus

FIGURE 33: Ici, le robot a détecté trois objets, il va donc effectuer l'exercice 3

Nous avons donc abandonné le module de détection de landmarks au profit d'une solution plus simple. Nous avons demandé au robot de détecter les objets bleus au sol, devant lui. Avec cette méthode, nous pouvons obtenir un résultat semblable : le robot devra simplement s'arrêter dès qu'il détecte les cubes bleus, compter le nombre de régions bleues détectées avec les outils d'imagerie que nous avons développés, et effectuer un exercice en fonction du résultat (par exemple, le numéro de l'exercice peut correspondre au nombre d'objets bleus devant lui).

5 Conclusion

Ce rapport aborde la problématique du déplacement automatique du robot androïde Nao. L'implémentation du parcours VITA nous a servi de prétexte pour l'exploration des capacités du Nao. De multiples facettes de ce robot ont été testés, avec plus ou moins de réussite. Seules celles qui nous ont satisfaites ont été retenues pour la version finale du programme.

Les choix d'implémentation effectués ont été guidés par les nombreuses expériences infructueuses et nous ont amené à ré-évaluer, en partie, le cahier des charges initial. L'obstacle majeur du projet a été la gestion de la vision embarquée du robot. Le délai de transmission élevé nous a notamment forcé à utiliser des images de faible qualité. L'incapacité à exploiter un flot d'images continu a également été une limitation importante. Vu l'importance de cette ressource au sein de notre projet, certaines fonctionnalités ont été compromises. Notons cependant que le Nao supporte en théorie l'acquisition d'un flux vidéo, du moment que cette fonctionnalité est implémentée en C++ et de manière embarquée.

Malgré les difficultés rencontrées, les fonctionnalités principales ont été réalisées avec succès. Cette expérience a présenté deux aspects intéressants. Premièrement cela nous a permis d'expérimentés les problématiques des systèmes embarqués, de leur gestion d'énergie, du calcul des trajectoires et des inter-communications. Ensuite, le point de vue de la robotique humanoïde nous a amenés à considérer des problèmes particuliers, absents de la robotique classique.

Références

[1] Aldebaran. Documentation officielle du nao (fournie sur le cd-rom).

[2] François Amato Bruno Barbieri. Cybernétique appliquée - rapport de projet, 2010.

[3] http ://effbot.org/zone/pil-histogram equalization.htm.

[4] http ://mail.python.org/pipermail/image-sig/2002 July/001907.html.

[5] http ://snipplr.com/view/22482/bresenhams-line algorithm/.

Table des matières

Table des figures