

Filière Systèmes industriels

Orientation Infotronics

Diplôme 2012

Jérémie Rappaz

*Développement d'un
démonstrateur pour l'Idiap
sur la base du robot NAO*

Professeur

Pierre Roduit

Expert

Flavio Tarsetti

SI	TV
X	X

<input checked="" type="checkbox"/> FSI <input type="checkbox"/> FTV	Année académique / Studienjahr 2011/2012	No TD / Nr. DA it/2012/27
Mandant / Auftraggeber <input type="checkbox"/> HES—SO Valais <input type="checkbox"/> Industrie <input checked="" type="checkbox"/> Etablissement partenaire Partnerinstitution IDIAP	Etudiant / Student Jérémie Rappaz Professeur / Dozent Pierre Roduit	Lieu d'exécution / Ausführungsort <input type="checkbox"/> HES—SO Valais <input type="checkbox"/> Industrie <input checked="" type="checkbox"/> Etablissement partenaire Partnerinstitution
Travail confidentiel / vertrauliche Arbeit <input type="checkbox"/> oui / ja ¹ <input checked="" type="checkbox"/> non / nein	Expert / Experte (données complètes) Flavio Tarsetti IDIAP Avenue des Prés Beudin 20 1920 Martigny	

Titre / Titel

Développement d'un démonstrateur pour l'Idiap sur la base du robot NAO

Description et Objectifs / Beschreibung und Ziele

L'Institut de Recherche Idiap (<http://www.idiap.ch>) spécialisé dans les interfaces homme-machine et le traitement de l'information multimédia continue de développer des démonstrateurs technologiques, afin de présenter l'institut au grand public.

Le projet de semestre ayant permis de démontrer que le robot NAO (<http://www.aldebaran-robotics.com>) pouvait facilement être utilisé comme objet de démonstration (reproduction de mouvements humains acquis par une Kinect), l'objectif du projet de diplôme sera de poursuivre avec la réalisation d'un démonstrateur complet basé sur ce robot.

Ce projet de diplôme sera séparé en 2 phases. Lors de la première, l'objectif sera de commander le robot par la voix. Une application devra être implémentée – basée sur une librairie existante de l'Idiap – et devra permettre de reconnaître les ordres donnés au robot et d'exécuter ceux-ci (e.g. "tourne à droite" devra provoquer une rotation du robot). Il faudra donc lister tous les scénarios (actions réalisables par le robot) et créer une application de commande qui sera capable de passer d'un scénario à l'autre. La stabilité du démonstrateur devra aussi être optimisée, afin que ce démonstrateur soit capable de fonctionner correctement avec plusieurs "interlocuteurs" et durant une durée importante.

Lors de la deuxième phase, qui sera exploratoire, un algorithme de *head pose estimation* sera porté sur la plateforme NAO, afin que le robot puisse reconnaître l'orientation de la tête de son interlocuteur et reproduire cette posture.

Objectifs :

- Portage d'un algorithme de commande vocale sur la plateforme NAO
- Implémentation des différents scénarios d'interaction avec le robot et implémentation du système de commande
- Test et analyse de la stabilité du démonstrateur
- Portage d'une librairie de head pose estimation.

Délais / Termine

 Attribution du thème / Ausgabe des Auftrags:
 14.05.2012

 Exposition publique / Ausstellung Diplomarbeiten:
 31.08.2012

 Remise du rapport / Abgabe des Schlussberichts:
 09.07.2012 | 12h00

 Défense orale / Mündliche Verteidigung:
 Semaine / Woche 36

Signature ou visa / Unterschrift oder Visum

 Responsable de l'orientation
 Leiter der Vertiefungsrichtung:

 Etudiant/Student:¹

¹ Par sa signature, l'étudiant-e s'engage à respecter strictement la directive et le caractère confidentiel du travail de diplôme qui lui est confié et des informations mises à sa disposition.
Durch seine Unterschrift verpflichtet sich der Student, die Richtlinie einzuhalten sowie die Vertraulichkeit der Diplomarbeit und der dafür zur Verfügung gestellten Informationen zu wahren.



Développement d'un démonstrateur pour l'Idiap sur la base du robot Nao



Diplômant/e

Jérémie Rappaz

Objectif du projet

L'objectif de ce projet a été d'imaginer et de concevoir un logiciel de démonstration permettant d'exposer le fruit des travaux de recherche de l'IDIAP. Cette démonstration fait l'utilisation du robot Nao d'Aldebaran.

Méthodes | Expériences | Résultats

Ce travail de diplôme consiste principalement à l'implémentation d'un logiciel faisant utilisation d'algorithmes développés à l'IDIAP. Ces algorithmes traitent principalement d'interface homme-machine et de biométrie comme par exemple la reconnaissance de la parole ou du visage.

Ce projet a connu deux phases d'implémentation. La première a permis à Nao de réagir aux commandes vocales d'un utilisateur grâce au logiciel *Juicer*, développé à l'IDIAP. Nao est désormais capable d'associer des commandes vocales à des actions programmables par l'utilisateur. Ces actions sont développées sous la forme de modules python et permettent donc de modifier facilement le contenu d'une présentation.

La seconde partie du projet fait utilisation de la technologie *KeyLemon*. Celle-ci permet à Nao, via ses caméras, de connaître la position de l'utilisateur par rapport à sa propre position et de tourner la tête vers lui. Nao pourra ensuite « apprendre » le visage de l'utilisateur et s'en rappeler. Il reconnaîtra ensuite les utilisateurs contenus dans sa base de données.

Travail de diplôme | édition 2012 |



Filière

Systèmes industriels

Domaine d'application

Infotronics

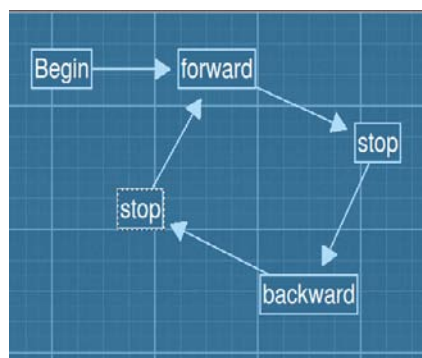
Professeur responsable

Dr. Pierre Roduit

pierre.roduit@hevs.ch

Partenaire

IDIAP research institute





Développement d'un démonstrateur pour l'Idiap sur la base du robot Nao



Diplômant/e Jérémie Rappaz

Objectif du projet

L'objectif de ce projet a été d'imaginer et de concevoir un logiciel de démonstration permettant d'exposer le fruit des travaux de recherche de l'IDIAP. Cette démonstration fait l'utilisation du robot Nao d'Aldebaran.

Méthodes | Expériences | Résultats

Ce travail de diplôme consiste principalement à l'implémentation d'un logiciel utilisant certains des algorithmes développés à l'IDIAP. Ces algorithmes sont orientés vers des domaines tel que l'interface homme-machine et la biométrie. Nous pouvons citer, comme exemple de leurs travaux, la reconnaissance de la parole et la reconnaissance du visage.

Ce projet a connu deux phases d'implémentation. La première a permis à Nao de réagir aux commandes vocales d'un utilisateur grâce au logiciel *Juicer*, développé à l'IDIAP. Nao est désormais capable d'associer des commandes vocales à des actions programmables par l'utilisateur. Ces actions sont développées sous la forme de modules python et permettent donc de créer et modifier facilement une présentation.

La seconde partie du projet fait utilisation de la technologie *KeyLemon*. Celle-ci permet à Nao, via ses caméras, de connaître la position de l'utilisateur par rapport à sa propre position et de tourner la tête vers lui. Nao pourra ensuite « apprendre » le visage de l'utilisateur et s'en rappeler. Il reconnaîtra ensuite les utilisateurs contenus dans sa base de données.

Travail de diplôme | édition 2012 |

Filière

Systèmes industriels

Domaine d'application

Infotronics

Professeur responsable

Dr. Pierre Roduit

pierre.roduit@hevs.ch

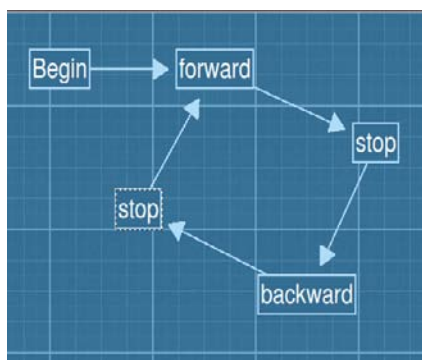
Partenaire

Idiap Research Institute

Responsable

M. Flavio Tarsetti

flavio.tarsetti@idiap.ch



Séquence d'actions créée par l'utilisateur en vue de sa démonstration. Le robot réagira à la voix selon cette séquence.

Nao reconnaît un utilisateur qu'il a déjà vu auparavant. En spécifiant un nom via le logiciel, le robot peut vous saluer à votre arrivée !

Référants

Dr. Pierre Roduit - HEVs

M. Flavio Taretto - IDIAP



Table des matières

1. Introduction	1
2. ASR	6
2.1. Survol	6
2.2. Juicer	6
2.3. Dépendances	7
2.4. Construction automatique de WFST	7
2.5. Configuration de Juicer	8
2.6. Création du <i>state chart</i>	9
2.7. Séquence de configuration	12
3. Application principale	13
3.1. Survol	13
3.2. Structure logicielle	13
3.3. Serveur	14
3.4. Côté client	14
3.5. Côté serveur	16
3.6. Base de données	17
3.7. Modules	18
3.8. Lanceur de modules	18
3.9. GUI	19
3.10. Synchronisation entre la vue et le modèle	21
3.11. Coordination des modules	21
3.12. Module kinect	23
3.13. Sources de flux audio	23
3.14. Script d'exécution	24
3.15. Discussion des résultats	24
4. Face recognition	26
4.1. Survol	26
4.2. Structure	26
4.3. Communication avec Nao	26
4.4. Séquencement des actions	27
4.5. Détection de visages	28
4.6. Reconnaissance faciale	28
4.7. Gestion des modèles et affichage des informations utilisateur	29

4.8. Utilisation de Nao	30
4.9. Formats d'images	33
4.10. Discussion des résultats	34
5. Conclusion	37
5.1. Discussion des résultats	37
5.2. Conclusion	37
Bibliographie	39
6. Annexes	40
A. Séquence de configuration de <i>Juicer</i>	41
B. Détail de l'interface graphique	43
C. Guide : configuration de la démonstration	45
D. Guide : mise en place de la démonstration	47
E. Liste des modules python déjà disponibles	51

1. Introduction

Le présent travail constitue le projet de fin d'étude à la HES-SO lors du semestre de printemps 2012. Il résulte d'un partenariat entre l'HES-SO Valais et l'Idiap Research Institute. Ce document expose le développement et les résultats obtenus lors de ce projet.

L'Idiap¹ est un institut de recherche à buts non-lucratifs spécialisé dans le traitement d'informations multimédias et dans l'interaction homme-machine. Dans le but d'exposer ses travaux de recherche, l'institut a été équipé d'une *show-room* destinée aux journalistes et aux visiteurs. Celle-ci dispose déjà de nombreux démonstrateurs permettant de mettre en scène les travaux des diverses branches de recherche. Ce projet vise à étoffer la *show-room* d'un démonstrateur supplémentaire en faisant intervenir le robot Nao.

1. IDIAP Research Institute. <http://www.idiap.ch>

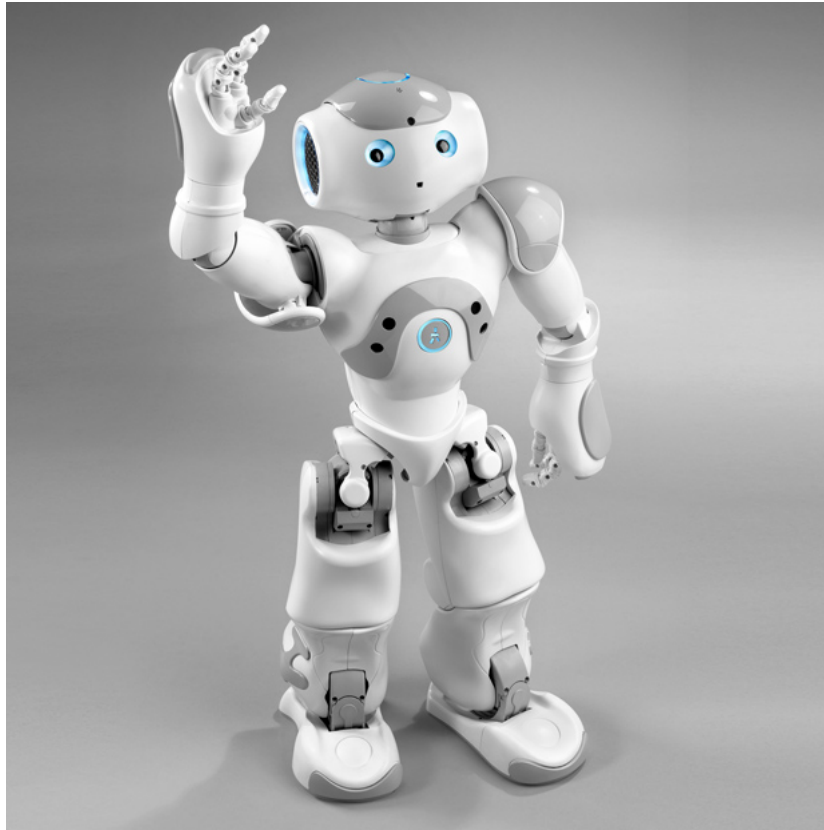


Figure 1.1.: Nao, le robot conçu par Aldebaran.

Nao est un robot humanoïde développé par Aldebaran². Celui-ci a été conçu pour la recherche et le développement et possède donc une gamme aussi large que diverse de fonctionnalités (nous ne les détaillerons d'ailleurs pas toutes ici). Nao ne possède pas moins de vingt-cinq degrés de liberté. Il possède également deux caméras, quatre microphones, un sonar et un capteur inertiel. Il embarque un processeur Intel Atom hébergeant un noyau Linux. L'Idiap s'est récemment équipée de plusieurs de ces robots et espère donc pouvoir marier les nombreuses capacités de celui-ci avec le fruit de ses recherches.

L'Idiap possède un secteur dont les recherches se focalisent sur la reconnaissance de la parole. Les chercheurs de ce secteur ont mis au point une librairie, nommée *Juicer*³, en partenariat avec les universités de Sheffield et d'Edimbourg. Celle-ci fonctionne sur le principe d'un ASR ou *Automatic Speech Recognition*. En d'autres termes, elle permet d'extraire d'un signal audio des mots ou des phrases prédéfinis prononcés par un utilisateur. A partir de là, nous avons pu imaginer une démonstration dans laquelle Nao exécuterait diverses actions en fonction des ordres vocaux intimes par l'utilisateur.

2. <http://www.aldebaran-robotics.com/>

3. Page web du projet : <http://juicer.amiproject.org/juicer/>. Notons que la librairie est disponible en *open source*.

Durant le déroulement du projet, nous avons été motivé par l'idée d'obtenir une démonstration fonctionnelle, facile à utiliser et surtout flexible dans sa configuration. C'est ce besoin de flexibilité qui nous a poussé à concevoir les actions de Nao sous la forme de modules. Chaque module se trouve sous la forme d'un script, correspondant à une action unique de Nao. Ces scripts sont relativement faciles à concevoir et ne requièrent que peu de connaissances techniques préalables pour être appréhendés. Ce mode de fonctionnement ne trouve donc ses limites que dans les possibilités physiques du robot et permet à l'utilisateur de se concentrer sur les aspects créatifs de la démonstration.

La seconde partie de ce travail s'est portée sur un autre champ de recherche de l'Idiap : la reconnaissance faciale. Les recherches de ce secteur ont donné naissance à une technologie de localisation et de reconnaissance des visages du nom de *KeyLemon*. Notons que cette technologie est commercialisée par la société du même nom⁴ et permet à un utilisateur de se connecter à son ordinateur par la reconnaissance de ses caractéristiques biométriques.

Chacun de ces modules va ensuite pouvoir être associé à une commande vocale. L'action contenue dans le module sera exécutée lors de la réception de cette commande. Nous avons aussi conçu un système permettant d'établir un cheminement entre ces modules sous la forme d'un diagramme d'état. Cela permet de fixer une logique propre à la démonstration. La conception de ce diagramme s'effectue directement dans l'interface graphique et apporte un contrôle visuel de l'action en cours⁵.

La seconde partie de ce travail s'est donc porté sur la conception d'un module faisant intervenir la technologie *KeyLemon*. Nous avons tout d'abord pu exploiter l'algorithme de détection des visages dans une image pour permettre à Nao de tourner la tête dans la direction d'un utilisateur présent dans son champ de vision. Nous avons ensuite tiré profit de l'algorithme de reconnaissance des visages afin que Nao puisse enregistrer le visage d'un nouvel utilisateur et que celui-ci puisse le reconnaître dans le futur. Si l'utilisateur entre son nom dans le logiciel Nao peut, par la suite, saluer cet utilisateur lorsqu'il le voit.

4. Page web de *KeyLemon* : www.keylemon.com

5. Annexe n°2

Nous allons maintenant exposer une représentation simplifiée de l'architecture de notre application afin d'avoir une vision de celle-ci dans sa globalité.

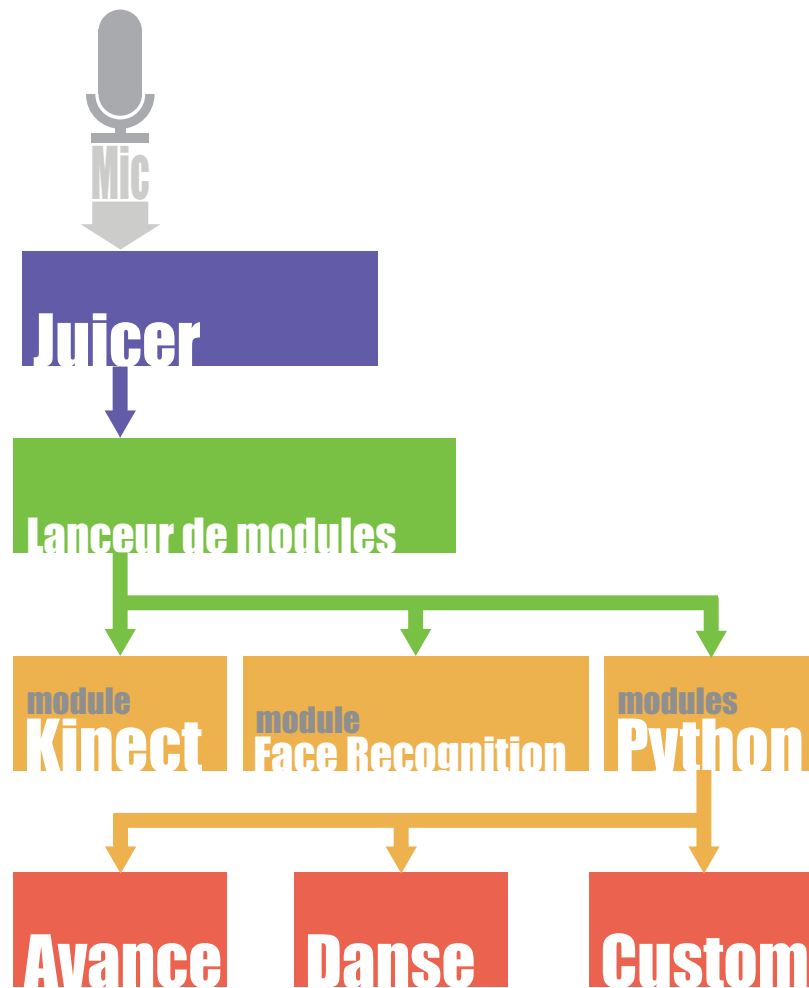


Figure 1.2.: Représentation simplifiée du fonctionnement de l'application.

Nous pouvons observer sur cette figure la façon dont le robot est contrôlé par notre application. Tout d'abord *Juicer* traite le signal émis par le microphone afin d'en extraire les commandes vocales potentielles. Lorsque *Juicer* détecte l'une desdites commandes, il transmet l'information à un bloc logiciel que nous nommerons "lanceur de modules". Celui-ci va s'occuper de faire correspondre la commande avec un module. Nous pouvons distinguer trois types de modules. Le premier concerne l'algorithme de *face recognition* que nous avons déjà abordé dans ce chapitre. Le second constitue l'ensemble des modules python. Nous pouvons voir certains des modules python déjà implémentés comme "avance" ou "danse". "Custom" symbolise ici les modules que l'utilisateur peut lui-même écrire. Le dernier type de modules est le module Kinect qui a fait l'objet du projet de semestre (PrS). Pour rappel, celui-ci permet à Nao de

copier les mouvements de l'utilisateur. Le fonctionnement de ce module ne sera pas abordé dans ce document et nous suggérons au lecteur de se référer au rapport du PrS pour plus de détails.

Pour une mise en route rapide du logiciel, le lecteur trouvera en annexe un guide de configuration⁶, ainsi qu'un guide de mise en place d'une démonstration⁷. Se trouve également en annexe une description complète de l'interface graphique⁸.

6. Annexe C

7. Annexe D

8. Annexe B

2. ASR

2.1. Survol

Ici, nous allons nous pencher sur la reconnaissance vocale, qui consiste en la partie maîtresse du présent travail. Nous allons y étudier le fonctionnement d'un ASR ainsi que son utilisation dans notre projet.

Un ASR, ou *Automatic Speech Recognition*, est une technique permettant d'analyser la parole au moyen d'un microphone et de la transcrire sous forme de texte. Nous allons donc exposer dans ce chapitre l'intégration d'un ASR au logiciel de contrôle de Nao afin de le piloter par la voix.

2.2. Juicer

Juicer est une librairie ASR développée à l'IDIAP qui permet de reconnaître des phrases prédéfinies sous la forme de *Weighted Finite State Transducer* (WFST). En d'autres termes, elle est capable de détecter des mots et d'effectuer le cheminement d'une phrase selon des liaisons prédéfinies entre ses mots. Le modèle de décodage est par conséquent considéré comme statique.

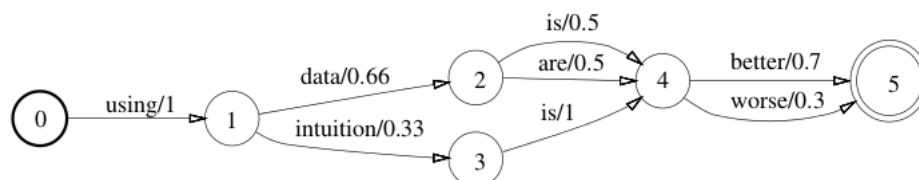


Figure 2.1.: Exemple de WFST. Le cheminement entre les mots est prédéfini. Chaque transition est accompagnée d'une certaine probabilité permettant de pondérer les chances de déplacement d'un mot à un autre.

Afin de pouvoir construire un modèle acoustique, les chercheurs de l'IDIAP ont établi un dictionnaire à l'aide d'enregistrements du Grand Conseil. Pour se faire une idée de son ampleur, ce dictionnaire possède plus de 12'000 entrées, chacune possédant un modèle acoustique équivalent.

Il faut distinguer plusieurs types de modèles au sein de cette architecture :

- Les modèles de lexiques : connaissance des mots et de leur prononciation, peut inclure plusieurs prononciations par mot.
- Les modèles phonétiques : représentent les unités fondamentales du lexicon de prononciation.
- Les modèles acoustiques : représentent la densité de probabilité d'émission de phonèmes selon le contexte.
- Les modèles de langages : représentent des phrases ou des commandes, ils dépendent des caractéristiques acoustiques de l'utilisateur.

–

Nous n'allons considérer, dans le présent travail, que les modèles de langages et les modèles de lexiques. Nous allons constituer les premiers à l'aide des seconds. Les modèles phonétiques et acoustiques ne nous apparaîtront qu'à titre indicatif dans les scores obtenus par Juicer.

2.3. Dépendances

Juicer utilise plusieurs librairies dont nous allons brièvement détailler les éléments principaux.

- Pulse audio : librairie permettant l'interfaçage avec les flux audio hardware.
- torch3 : librairie de *statistical machine learning*.
- tracter : librairie de *speech signal processing*.
- librosample : librairie permettant le *resampling* (passage d'une fréquence de sampling à une autre).
- kissFFT : librairie de *Fast Fourier Transform*.
- srilm : Toolkit permettant la construction de modèles de langages statistiques.

2.4. Construction automatique de WFST

La première étape du travail a été d'automatiser la construction de modèles de langages. L'écriture d'un script en python a donc permis de générer dynamiquement l'établissement des différents chemins possibles de la WFST. L'utilisateur est invité à entrer les commandes qu'il désire fixer comme déclencheurs d'actions du robot et le script a pour fonction d'en établir le *state chart*. Selon le principe de fonctionnement de Juicer, ce diagramme comprend un point d'entrée et un point de sortie. Entre ces deux points sont représentées toutes les possibilités de phrases potentielles. Il est à noter que selon son principe de fonctionnement, Juicer traite l'information de bruit comme si celle-ci était un état possédant toutes les phonèmes en tant que *trigger*. Cet état de bruit est rebouclé sur l'état initial. Nous oscillons donc entre ces deux états jusqu'à détecter le début d'une phrase potentielle.

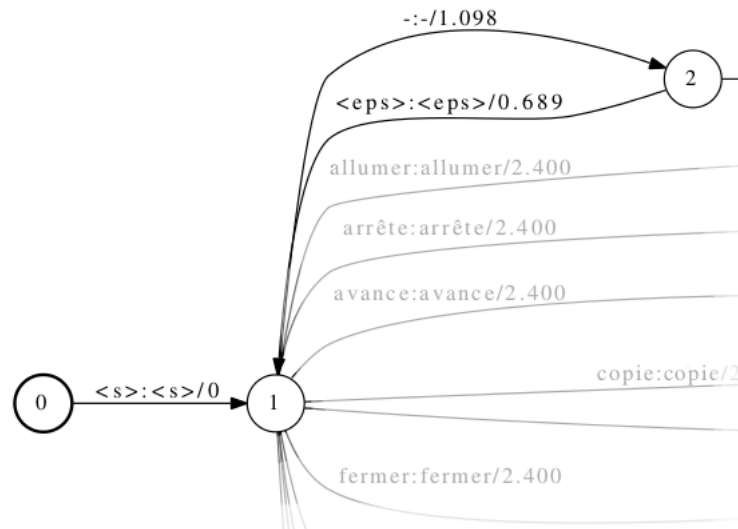


Figure 2.2.: Représentation du bruit dans le “*state chart*” WFST de Juicer.

Chaque transition possède une probabilité de s’effectuer calculée en fonction du nombre de transitions potentielles depuis l’état considéré.

$$P(x) = \frac{1}{N}$$

Où $P(x)$ représente la probabilité de la transition x lorsque N est le nombre total de transitions possibles depuis l’état considéré.

2.5. Configuration de Juicer

Le script permet de configurer à la fois Juicer mais aussi l’application qui sera décrite dans le chapitre suivant.

Tout d’abord, celui-ci génère 4 fichiers nécessaires à la configuration de Juicer :

- utterances : contient les commandes à détecter.
- dictionary : contient les mots à détecter.
- WFST : contient les modèles acoustiques des mots nécessaires.
- G.fsm : contient le modèle WFST où chaque mot est défini par : un état le précédant, un état le suivant, l’index du mot dans le dictionnaire ainsi que sa probabilité de transition.

Le fichier G.fsm que nous voulons produire représente le *state chart* déterminé par les commandes entrées par l’utilisateur. C’est un fichier de texte qui se construit comme suit :

Chaque ligne correspond à un mot. Les mots à reconnaître sont représentés dans ce modèle comme étant des transitions entre deux états. Chaque ligne du fichier

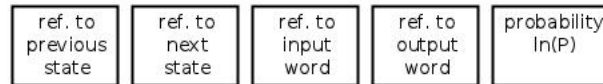


Figure 2.3.: Construction du fichier de configuration de Juicer.

représente donc à une transition. Les deux premières entrées de la ligne sont des identifiants uniques faisant référence à un état. La première correspond à l'état précédent la transition, tandis que la seconde correspond à l'état la suivant. Ces états sont définis par un index, lui-même défini par le script. Cet index est provisoire et sera redéfini par Juicer dans son modèle interne.

Les deux entrées suivantes correspondent à la référence du mot à reconnaître. Ces entrées font référence aux fichiers *G.insyms* et *G.outsyms* produits par les scripts de configuration de Juicer. Dans notre cas, chaque transition (et donc chaque ligne du fichier) comprend un symbole d'entrée et un symbole de sortie. Nous pouvons nous représenter cela en concevant que pour un symbole en entrée, le système place le symbole correspondant en sortie. Ceux-ci sont cependant identiques. En effet, ce système tend à produire une sortie avec les symboles détectés à l'entrée.

Le script génère ensuite un fichier de correspondance entre une commande et une action à effectuer. Il ira lire les différentes actions possibles (sous forme de modules python) et demandera à l'utilisateur de les associer à chacune des commandes. Le fichier résultant se nomme *matching.txt* et est contenu dans le dossier "config" du logiciel principal.

2.6. Création du *state chart*

Dans cette partie nous allons développer plus en détails le fonctionnement de l'algorithme utilisé pour la création du *state chart*.

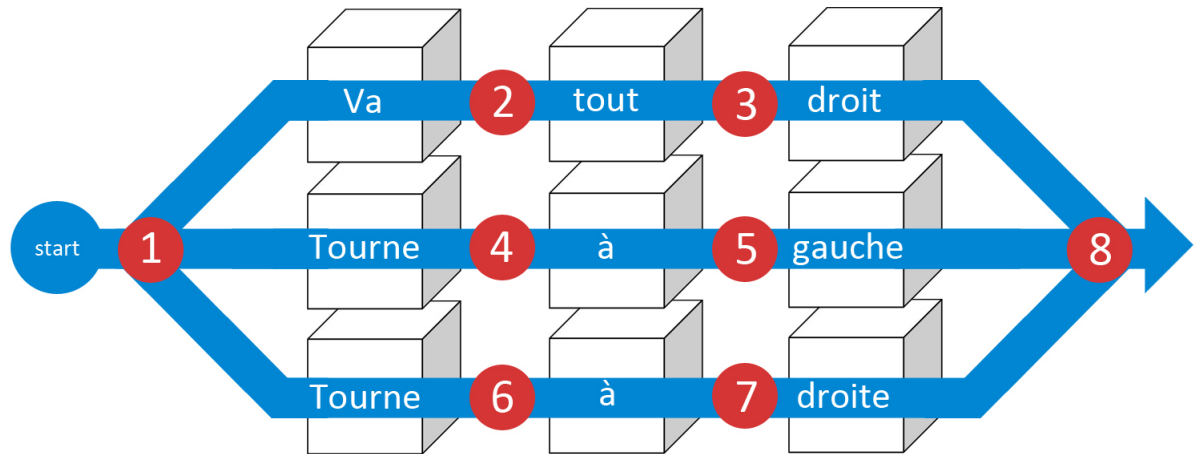


Figure 2.4.: Algorithme d'établissement du modèle de langage : établissement du *state chart* initial.

Nous ordonnons tout d'abord les commandes entrées en une matrice de deux dimensions. Notons que le script utilise des listes de listes, mais nous conserverons l'idée de la matrice pour simplifier l'explication. La première dimension contient les commandes (à l'horizontale sur la figure) alors que la seconde contient les mots qui y sont contenus (à la verticale sur la figure). Chaque mot de chaque commande est représenté sous la forme d'une liste contenant les informations qui lui sont relatives (voir le chapitre "configuration de Juicer") ainsi qu'une booléenne permettant de déterminer le mot comme étant un passage potentiel. Nous illustrons cette liste relative au mot par un cube sur la figure ci-dessus.

La première opération va permettre de fixer les bons indexes relatifs aux états du *state chart*. Il est important de noter que ce sont les états et non pas les mots qui possèdent un index (rappelons que les mots sont des transitions entre les états dans ce modèle).

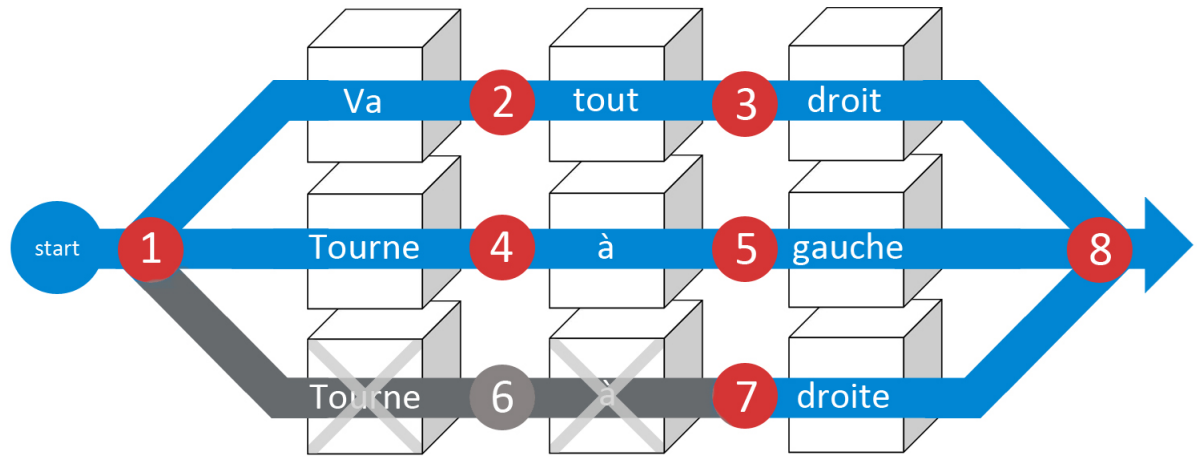


Figure 2.5.: Algorithme d'établissement du modèle de langage : suppression des chemins superflus. Ici, l'algorithme a déterminé qu'il existait plusieurs occurrences du même mot dans la première et la deuxième colonne. Il désactive donc les mots superflus. Le passage par ces mots dans le *state chart* est maintenant rendu impossible.

L'algorithme va ensuite effectuer plusieurs passages sur la matrice afin d'éliminer les transitions superflues. Lors du premier passage, l'algorithme va parcourir chaque colonne de la matrice afin de déterminer si un mot possède plusieurs occurrences. Pour faciliter cela, nous avons préalablement ordonné les commandes dans l'ordre alphabétique. Si plusieurs occurrences sont détectées, toutes les occurrences suivant la première sont "désactivées" en fixant la booléenne du mot à zéro.

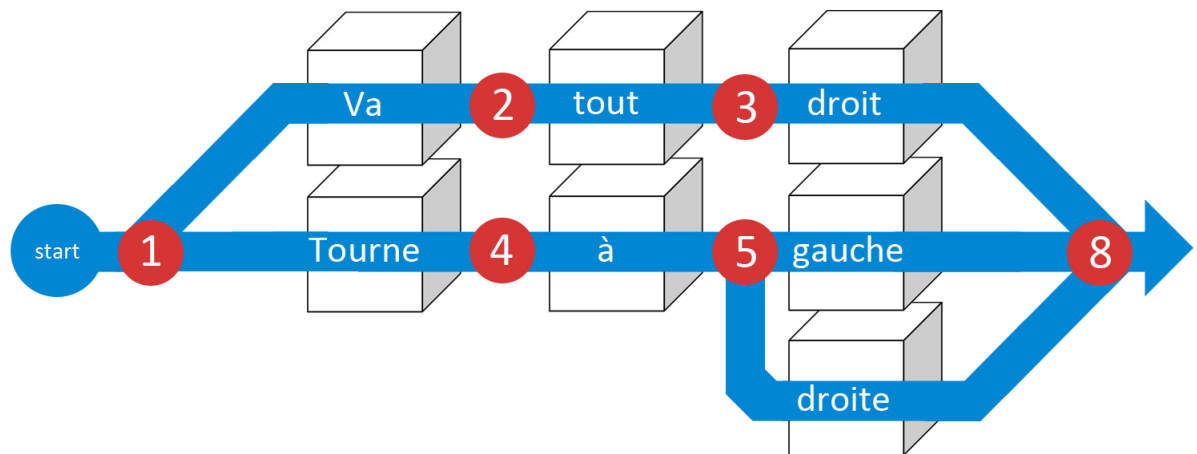


Figure 2.6.: Algorithme d'établissement du modèle de langage : reconfiguration du *state chart* après la suppression des chemins superflus.

Une fois que tous les chemins potentiels superflus ont été désactivés, l'algorithme doit reconfigurer les parties restantes de ces chemins. Cela s'avère possible en récupérant l'index de l'état dont part le premier mot à ne pas être redondant dans le premier des chemins concernés. Cet index est ensuite fixé en tant qu'index de départ des premiers mots à ne pas être redondants dans les chemins concernés.

Il est plus aisé de comprendre cela avec l'exemple de la figure ci-dessus. Ici, dans le premier des chemins concernés, le chemin 1-4-5-8, le premier mot à ne pas être redondant est le mot "gauche". L'état dont part ce mot est l'état n°5. On fixe donc l'état n°5 comme état de départ du mot "droite".

L'algorithme se doit ensuite de faire un dernier passage à travers la matrice afin de déterminer les probabilités aux embranchements (voir plus haut à "Construction automatique de WFST") et de les stocker dans les informations de chaque transition. Prenons l'exemple de la figure. A l'état n°5, il y a 50% de chance que le prochain mot soit "gauche" et 50% de chance que ce dernier soit "droite". Cette probabilité donne donc environ 0.693 sous sa forme logarithmique. Cette valeur est donc stockée dans la liste faisant référence au mot "gauche" ainsi que dans celle faisant référence au mot "droite".

Une fois arrivé à ce stade, nous avons une matrice ayant la bonne forme. Il faut cependant encore l'écrire sous forme de fichier. Nous devons écrire dans le fichier G.fsm. Dans ce fichier, chaque ligne correspond à une transition et donc à un mot. Les mots étaient jusqu'ici considérés comme des chaînes de caractères. Elles sont désormais remplacées par un chiffre faisant référence aux index générés par Juicer. Ceux-ci sont contenus dans les fichiers G.insyms et G.outsyms.

2.7. Séquence de configuration

Un schéma détaillant la séquence de configuration de Juicer est disponible en annexe¹.

1. Annexe A

3. Application principale

3.1. Survol

Nous allons maintenant décrire l'application de contrôle de Nao. Cette application a pour tâche principale de recevoir des commandes vocales via Juicer et de lancer des modules d'action correspondants.

L'application comprend deux types d'actions qui pourront être envoyées au robot. Le premier type permet le lancement de modules évolués, codés en C++ et par conséquent n'offre que peu de flexibilité. Un bon exemple de ce type de modules est l'application de téléopération via la Kinect ¹.

L'objectif du deuxième type de modules est d'obtenir une application permettant une certaine modularité. En effet, elle permet d'exécuter des script python qui pourront être facilement écrits et modifiés.

Enfin, l'application permet d'ordonner des actions sous la forme de *state chart* au sein d'une GUI. En définissant les transitions possibles, l'application offre à l'utilisateur la possibilité de structurer ses actions et d'éviter des transitions non désirées.

3.2. Structure logicielle

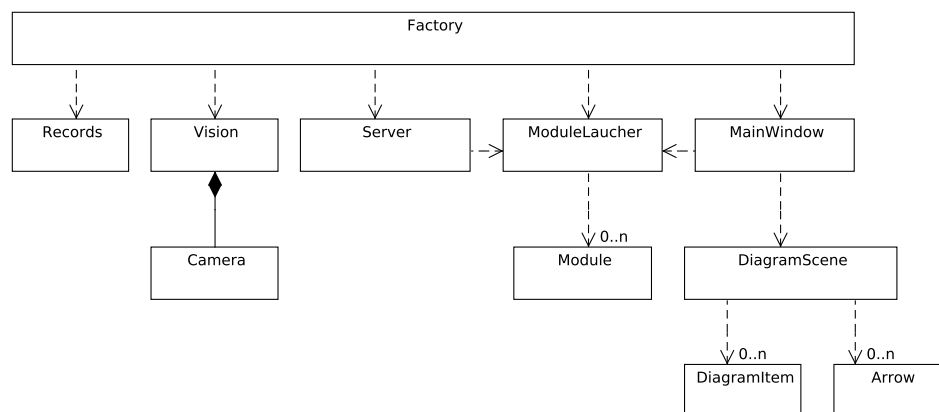


Figure 3.1.: Structure de l'application principale.

1. cf. Rapport du PrS

La structure décrite ici comporte plusieurs parties distinctes :

- Serveur : s’occupe de la communication avec Juicer.
- Lanceur de modules : permet de séquencer et lancer les différents modules.
- GUI : permet de concevoir de manière graphique les chemins possibles entre les différents modules (représentée sur cette figure par la *MainWindow*, une classe spécifique au *framework Qt*).
- Records : enregistre les résultats obtenus par Juicer dans une base de donnée SQLite.

3.3. Serveur

Le serveur s’occupe essentiellement d’établir la communication avec Juicer. L’application utilise pour cela un *socket*, méthode de communication inter-logiciels relativement simple à implémenter. Le logiciel Juicer a été recompilé avec un client TCP minimaliste, permettant de se connecter sur un serveur en *local host* sur un port défini. Il transmet ensuite ses résultats sous la forme d’une chaîne de caractères à chaque fois qu’une commande est détectée.

Le *socket* utilisé côté client provient de la librairie de *socket UNIX*, accessible via “*socket.h*”. Cette librairie est simple, bien documentée et ne rajoute pas de dépendances supplémentaires à la compilation de Juicer.

De son côté, la classe *Server* crée un serveur à l’écoute sur le bon port et récupère les données dès qu’une connexion est établie.

Notons qu’à chaque fois qu’une commande est détectée, une connexion est établie, une transmission s’effectue, puis la connexion est terminée par le client. Une alternative aurait été de conserver une connexion active en permanence, mais cela aurait demandé l’implémentation d’un mécanisme pour déterminer si le client est toujours connecté (“*keep alive*”). Cela n’est pas nécessaire avec cette implémentation.

On peut aussi noter que ce type d’implémentation n’est pas *thread safe*. Cependant, nous sommes ici dans un cas de connexion unique et il n’arrivera jamais que plusieurs clients se connectent en même temps sur le même serveur. Ce mécanisme n’est donc pas non plus nécessaire.

3.4. Côté client

Nous pouvons voir ici la manière dont nous avons rajouté une interface réseau à la librairie Juicer. Celle-ci permettra de communiquer ses résultats via un *socket*. La librairie a été recompilée intégralement après cette modification.

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
connect(sockfd, (sockaddr*)&serv_addr, sizeof(serv_addr));
n = write(sockfd, toSend, strlen( toSend ));
close(sockfd);
```

Figure 3.2.: Code source : connexion côté client.

Nous pouvons observer ici le code simplifié nécessaire à l’envoi de données côté client. Celui-ci crée un *socket* en ligne 1. Les paramètres sont respectivement : famille, type protocole².

- Famille : représente la famille de protocole utilisé. `AF_INET` pour TCP/IP utilisant une adresse Internet sur 4 octets.
- Type : indique le type de services (orientés connexion ou non). Dans le cas d’un service orienté connexion (ce qui est généralement le cas), l’argument type doit prendre la valeur `SOCK_STREAM` (communication par flot de données). Dans le cas contraire (protocole UDP), le paramètre type doit alors valoir `SOCK_DGRAM` (utilisation de datagrammes, blocs de données).
- Protocole : permet de spécifier un protocole permettant de fournir le service désiré. Dans le cas de la suite TCP/IP il n’est pas utile, nous le mettrons ainsi toujours à 0.

Le *socket* se connecte ensuite à l’adresse du serveur distant à la ligne 2. Dans notre cas, nous communiquons en *loopback*. Afin de déterminer l’adresse de destination de manière dynamique, nous pouvons utiliser la fonction ci-dessous :

```
server = gethostbyname("localhost");
```

Ensuite, et seulement si la connexion est établie (voir plus loin), nous pouvons commencer à “écrire” à travers le *socket* via la méthode *write* visible à la ligne 3. Celle-ci prend respectivement comme paramètres le *socket* à utiliser (celui-ci est en fait un *integer* représentant l’identifiant unique du *socket*), un pointeur sur un tableau de caractères ainsi que la longueur de celui-ci.

Enfin, la connexion est fermée via la méthode *close* visible ici en ligne 4.

Les données sont transmises sous la forme de chaînes de caractères. Il a donc fallu déterminer un format permettant de transmettre en une seule fois toutes les données correspondantes à une phrase détectée. Celles-ci sont constituées de :

- La phrase détectée, sans espaces, sous le format de Juicer, c’est-à-dire avec la représentation du silence au début et à la fin. Ex. : `<s>lèvelesbras</s>`
- Le score de modèle acoustique de la phrase détectée sous la forme d’un *float* d’au plus 4 chiffres après la virgule.

2. <http://www.commentcamarche.net/contents/sockets/sockfonc.php3>

- Le score de modèle de langage de la phrase détectée sous la forme d'un *float* d'au plus 4 chiffres après la virgule.

Ces données subissent donc une concaténation avec un caractère de délimitation “#”. Celui-ci à été choisi arbitrairement.

Il est à noter que l'établissement d'une connexion de ce type peut rapidement mettre en péril la stabilité de l'application. En effet, la création du *socket* étant tributaire d'un intervenant extérieur (ici le serveur), il est essentiel de bien formater les données provenant de celui-ci. Plusieurs contrôles sont donc nécessaires afin de garantir le bon fonctionnement de la connexion :

- Le *socket* à bien été “ouvert”
- L'hôte distance existe
- La connexion est bien établie

Si ces conditions sont bien remplies, nous pouvons tenter d'écrire dans le *socket*.

Nous pouvons aussi noter que nous utilisons ici la fonction *snprintf* qui permet un formatage des données sécurisé. En effet, cette méthode permet de préciser la taille du *buffer* de destination, ce qui permet d'éviter les débordements de capacité.

3.5. Côté serveur

Du côté du serveur, nous utilisons un ensemble de classes provenant du *framework* Qt qui nous facilite grandement la tâche en matière d'implémentation. Observons une partie du code utilisé.

```
tcp = new QTcpServer(this);
tcp->listen(QHostAddress::Any, 9876);
QObject::connect(tcp, SIGNAL(newConnection()), this, SLOT(printline()));
```

Figure 3.3.: Code source : connexion côté serveur.

Nous nous servons principalement de la classe *QTcpServeur*. Celle-ci a juste besoin d'être initialisée par l'appel à sa méthode *listen* tout en lui spécifiant une adresse et un port. Après la seule exécution de ces deux premières lignes, le serveur est déjà actif et à l'écoute.

Tout d'abord, à la création de la classe serveur, nous utilisons le mécanisme de “SIGNAL/SLOT” propre au *framework* Qt. Celui-ci nous permet d'appeler une fonction arbitraire à chaque émission du signal *newConnection()*. Celui-ci est bien sûr émis à chaque fois qu'un client initie une connexion à l'adresse du serveur en question.

Voyons maintenant comment nous traitons les données venant du client qui vient d’initier la connexion. Notons que cet extrait de code est simplifié et n’est présenté que dans le but d’illustrer notre propos.

```
QTcpSocket *so= tcp->nextPendingConnection();

so->waitForReadyRead(50);
QByteArray a=so->read(256);
if (a.endsWith("\r\n"))
    a.resize(a.length()-2);

QString data(a);
data.toAscii();

QStringList sl = data.split(QRegExp("\\#"));
QString ac = sl.at(1);
QString lm = sl.at(2);
```

Figure 3.4.: Code source : Réception des données côté serveur.

Comme dans l’établissement de la connexion côté client, nous devons créer un nouveau *socket* à chaque nouvelle connexion. Nous utilisons la classe *QTcpSocket*. Celle-ci est instanciée via la méthode *nextPendingConnection* qui nous retourne un *socket* déjà connecté. Nous appelons ensuite la méthode *waitForReadyRead*, qui permet d’attendre durant une certaine période l’arrivée des données. Il est à relever que cette méthode est bloquante. Elle a donc été fixée à 50ms afin de ne pas perturber le bon fonctionnement du logiciel. Notons que tous les aspects “temps réel” de ce logiciel (comme les classes permettant l’acquisition de données de la Kinect) ont été traités dans des *threads* séparés. Cette fonction ne met donc pas en péril leur bon fonctionnement.

Nous devons ensuite traiter les données émanant du client. Comme expliqué plus haut, celles-ci se trouvent sous la forme d’une chaîne de caractères. Nous devons en premier lieu nous assurer de l’absence de tout *carriage return* à la fin de la chaîne. Puis, nous pouvons séparer la chaîne en trois *String* en utilisant la fonction *split* à laquelle il suffit d’indiquer le caractère de séparation. Il ne reste plus alors qu’à faire retrouver à chaque paramètre sont type d’origine.

3.6. Base de données

Nous avons mis en place une base de donnée SQLite en *back-end*. Celle-ci a été créée dans le but de conserver une trace des scores du modèle de langage ainsi que ceux du modèle acoustique (il s’agit des modèles que nous retourne Juicer à chaque détection). Il es à noter que celle-ci s’enregistre par défaut dans un fichier unique “*.db” à la racine du fichier contenant l’application principale.

Cette base de données ne contient qu’une table dont voici les colonnes :

id INTEGER PRIMARY KEY	sentence VARCHAR(255)	ac FLOAT	lm FLOAT
------------------------	-----------------------	----------	----------

id : contient un index unique auto-incrémental sous la forme d’un entier.

sentence : contient la phrase détectée sous la forme d’une chaîne de caractères.

ac : contient le score du modèle acoustique sous la forme d’un *float*.

lm : contient le score du modèle de langage sous la forme d’un *float*.

A chaque fois que le résultat d’une détection nous parvient de Juicer via le serveur, nous créons une nouvelle entrée dans la table.

Nous affichons ensuite les résultats dans un panneau de l’interface graphique. Nous utilisons pour cela un mécanisme du *framework Qt*. Nousinstancions un objet de la classe *QSqlTableModel* et le lions à la table de notre base de données. Ce modèle sera ensuite mis à jour en même temps que cette table. Nous pouvons donc lier ce modèle à une *QTreeView* afin d’afficher en temps réel le contenu de la base de données dans l’interface graphique.

3.7. Modules

Nous allons exposer ici la représentation d’un module sous forme de classes. Celle-ci contient uniquement les attributs nécessaires à l’établissement de la relation entre les commandes transmises par Juicer, la représentation graphique des modules ainsi que le nom des fichiers python contenant les modules.

Ces attributs sont :

- *sentence* : phrase à laquelle le module doit réagir ; elle est construite avec le même format que celui provenant de Juicer.
- *func_name* : nom de la fonction à appeler.
- *file_name* : nom du fichier contenant le module.
- *unique_id* : identifiant unique permettant de faire le lien avec l’interface graphique.

3.8. Lanceur de modules

Cette classe logicielle a pour but de séquencer et de lancer des modules python. Pour cela, elle fait usage de “PythonQt”, une librairie prévue à cet effet. Elle permet d’embarquer des scripts python au sein d’une application C++ et d’y faire appel.

La façon d’implémenter les modules python a été standardisée dans le but de faciliter l’identification et l’utilisation des différents modules. Chaque module possède une

fonction unique dont le nom est le même que celui du fichier (nom du fichier + “.py”). Chacune de ces fonctions prend en paramètre l’adresse IP et le port où est connecté le robot. Chaque module inclut les dépendances dont il a besoin au début de la fonction (via *import*).

Nous allons maintenant observer plus en détail l’utilisation de la librairie “Python-Qt” et la façon dont celle-ci lance des modules.

```
PythonQtObjectPtr context =  
    PythonQt::self()->createModuleFromFile("somename",  
                                           path+str+".py");  
  
QVariantList args;  
args << ip;  
args << port;  
QVariant result = context.call(str, args);
```

Figure 3.5.: Code source : Appel à un module python.

Nous voyons ici l’utilisation l’appel d’un module python. Nous devons d’abord créer un “contexte”, c’est-à-dire un environnement python dans lequel nous importons un module. Cela est possible grâce à la méthode *createModuleFromFile* qui nous permet de charger le fichier spécifié comme le ferait la directive *import* de python. Nous donnons à ce module un nom arbitraire qui ne sera pas utile dans notre cas.

Nous pouvons ensuite, au sein de ce contexte, faire appel à une méthode grâce à la fonction *call* à laquelle nous spécifions le nom de la fonction ainsi que ses paramètres dans une liste. La fonction est ensuite appelée et retourne sa valeur comme elle le ferait en C++.

Ce mode de fonctionnement permet l’exécution de modules scriptés au sein d’une application compilée, et donc déterministe, ce qui offre une certaine flexibilité.

Notons qu’une série de modules a déjà été réalisée. Celle-ci comporte des actions basiques permettant de mettre en place une première démonstration. Une liste des modules existant se trouve en annexe³.

3.9. GUI

L’interface graphique à été conçue de façon à pouvoir définir l’ordre dans lequel les actions sont effectuées. Cela permet de définir les chemins possibles, mais aussi de limiter les possibilités de transitions à chaque état.

3. Annexe E

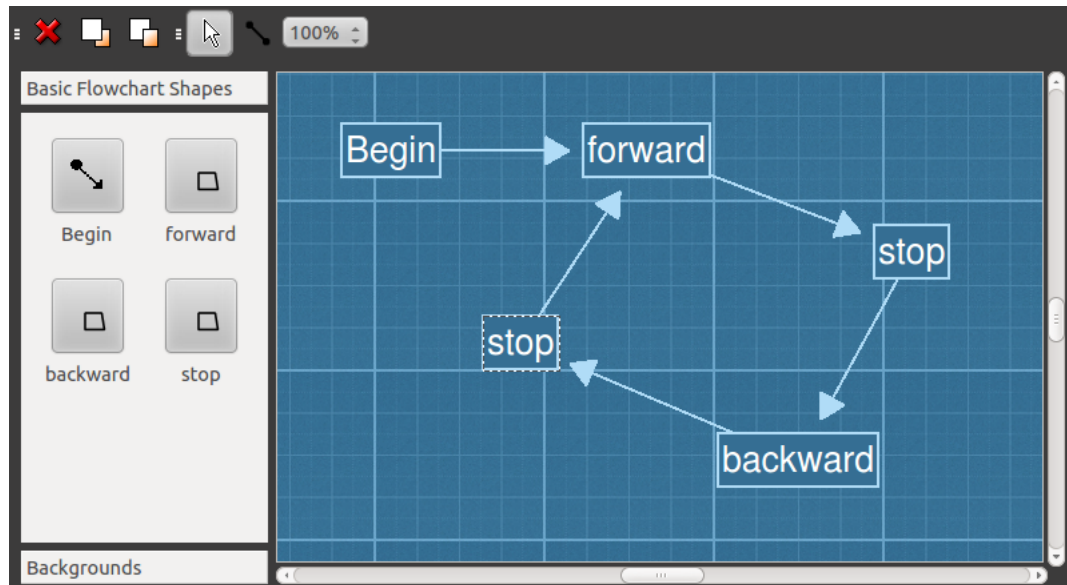


Figure 3.6.: Interface de l'application. Elle permet de séquencer les actions du robot à détecter. L'utilisateur n'a, pour cela, qu'à glisser les actions dans la zone d'édition et de dessiner les flèches nécessaires à l'établissement de son scénario.

La zone de travail de la GUI (en bleu sur la figure ci-dessus) permet d'importer des objets (de la classe *DiagramItem*) représentant des modules. Chaque objet se voit attribuer à sa création un identifiant unique permettant de le faire correspondre à son équivalent dans le modèle (selon une représentation MVC). A la création ou la destruction des modules dans l'interface graphique, un signal est émis afin d'actualiser le modèle. Nous conservons alors en permanence le modèle et la vue synchronisés.

L'interface graphique a été créée à partir d'un exemple fourni par Qt⁴. Celle-ci a été adaptée à aux besoins du présent projet. Notre travail a consisté à implémenter un mécanisme permettant de générer dynamiquement les éléments contenus dans la *toolbox* à gauche. Ceux-ci sont créés à partir des modules décrits dans le fichiers de configuration "matching.txt" défini au chapitre précédent. L'élément *begin*, quant à lui, permet de définir le point d'entrée du *state chart*.

Nous exposons ici la partie de l'interface graphique relative à ce chapitre. Pour une description complète, nous prions le lecteur de se référer au schéma en annexe⁵.

4. cf. *DiagramScene* dans les exemples de Qt Creator 2.4.1 utilisant le framework Qt dans sa version 4.7.4

5. Annexe B

3.10. Synchronisation entre la vue et le modèle

Nous allons traiter ici de la relation entre la vue et le modèle. A chaque fois qu'un utilisateur modifie le schéma de séquencement des modules via l'interface graphique, le modèle doit être mis à jour via un signal. Une liste *moduleOnView* est définie comme attribut de la classe *ModuleLauncher*. Cette liste contient un objet de la classe *Module* pour chaque module présent dans la vue. Ces objets contiennent les informations relatives aux modules (nom de fichier, nom de fonction...). La classe *ModuleLauncher* possède donc en tout temps un modèle du schéma que l'utilisateur constitue dans la vue.

Voici la liste des différents types de signaux qui permettent de garder le modèle à jour :

- Ajouter une flèche : le signal contient un pointeur sur le module suivant et un pointeur sur le module précédent la flèche; on fixe l'attribut *next* du premier module à l'adresse de second.
- Supprimer une flèche : le signal contient un pointeur sur le module suivant et un pointeur sur le module précédent la flèche; on fixe l'attribut *next* du premier élément à une valeur par défaut.
- Ajouter un module : le signal contient un pointeur sur le module inséré dans la vue; nous pouvons extraire de l'élément pointé un identifiant unique ainsi qu'un nom de module, un objet de la classe *Module* est instancié puis ajouté à la liste *ModuleOnView*.
- Supprimer un module : le signal contient un pointeur sur le module inséré dans la vue; nous pouvons extraire de l'élément pointé un identifiant unique ainsi qu'un nom de module; l'objet correspondant à cet identifiant est retiré de la liste *ModuleOnView*.

3.11. Coordination des modules

Chaque module qui est lancé par le logiciel est appelé *broker* (courtier en Français). Comme un courtier, celui-ci s'assure de la bonne transaction (ici de données) entre les deux parties. D'un point de vue plus pratique, on crée, en début de transaction, une instance de *ALMotionProxy*, à travers laquelle nous allons pouvoir communiquer nos consignes au robot. Nous devenons ainsi "client" du serveur de Naoqi (le système d'exploitation de Nao).

Il existe différents types d'ordres ou de consignes potentiels que nous pouvons transmettre au robot. Nous allons les détailler ici.

Certains ordres ne sont applicables qu'une seule fois, comme par exemple une consigne d'angle sur un joint précis du robot. Nous pouvons ainsi ordonner au robot de lever

le bras droit à quarante-cinq degrés, ordre qui aura une action unique⁶.

D'autres consignes permettent de lancer un processus au sein de Naoqi. Elles peuvent avoir une limite temporelle (ou une limite d'un type relatif à la nature de la commande) que nous lui fixons lors de leur création. Elles peuvent aussi être détruites, comme n'importe quelle tâche sur un système d'exploitation lambda. Pour donner un exemple plus parlant, nous pouvons imaginer donner au robot l'ordre de marcher en avant sur trois mètres⁷. Le robot s'arrêtera ensuite. Mais nous pouvons aussi ordonner au robot de commencer à marcher⁸ et de lui donner ensuite l'ordre de s'arrêter⁹ (et tenter ainsi de gérer au mieux l'arrêt du robot) ou même de "tuer" la tâche¹⁰ (représente un arrêt brutal du mouvement).

Il existe un dernier type de commandes que nous appellerons les configurations. Elles permettent d'agir sur les paramètres du robot et ainsi d'influer sur son comportement. La plupart de ces configurations servent à lancer des processus en arrière-plan sur le système d'exploitation du robot. Ils permettent d'influencer les futures commandes que nous passerons à Nao. Il existe, par exemple, une commande de configuration permettant de recalculer au mieux toutes les commandes futures passées au robot¹¹. Plus en détail, cette commande permet de déterminer si une consigne de mouvement serait susceptible de mettre en péril la stabilité du robot et la rectifie au mieux afin de la stabiliser.

Le problème qui est rapidement intervenu lors des premiers tests du logiciel provenait de l'existence de plusieurs "clients" connectés simultanément au serveur Naoqi. Lorsque chacun lance ses tâches ou effectue ses configurations, la situation devient rapidement problématique. Citons ici en exemple l'utilisation de la stabilisation des mouvements précitée, qui, salutaire dans le module de reproduction des mouvements, enraie le bon fonctionnement de la marche.

Afin de limiter ce genre de désagréments, nous avons fixé quelques règles essentielles à la téléopération de Nao via une architecture modulaire. Tout d'abord, chaque *broker* est créé à l'initialisation d'un module et détruit lors de la transition au module suivant. Cela permet de n'avoir, au plus, qu'un client connecté au serveur en même temps. Ensuite, nous limitons au maximum les configurations superflues et réinitialisons à leur valeur initiale les configurations effectuées dans chaque module lorsque ceux-ci ne sont plus utilisés. Il existe une exception à cela. En effet, le paramètre de tension appliquée sur les moteurs (ou *stiffness*) se doit de rester en permanence suffisamment élevé pour maintenir le robot dans la pose actuelle. Enfin, tout comme les configurations, les tâches lancées par un module se doivent d'être finies ou détruites lorsque le module est terminé par le logiciel. Ces quelques règles

6. cf. la méthode "setAngle"

7. cf. la méthode "walkTo"

8. cf. la méthode "setWalkTargetVelocity"

9. cf. la méthode "stopWalk"

10. cf. la méthode "killWalk"

11. cf. la méthode "wbEnable"

permettent une implémentation plus efficace et plus sûre lors de la création de nouveaux modules.

3.12. Module kinect

Nous allons traiter ici de l'intégration du module de téléopération via la kinect. Son fonctionnement et son développement ont déjà été abordés dans le projet de semestre. Nous prions donc le lecteur d'y faire référence pour tout l'aspect technique de cette partie.

Ce dont nous avons besoin, dans l'optique d'une implémentation de cette partie sous la forme d'un module, est de pouvoir démarrer et stopper la téléopération. A l'arrêt du module, le proxy de mouvement dont il se sert pour contrôler le robot doit être désactivé ou détruit afin de ne pas interférer avec les consignes des autres modules.

La seconde partie de l'intégration concerne l'interface graphique. Deux *QLabel* ont été ajoutés à la GUI afin d'afficher l'image de la caméra de la Kinect, ainsi que l'image représentant la profondeur de la *frame*. D'autres labels ont été ajoutés afin d'afficher des informations sur le *frame rate*, le temps écoulé etc.

Le module Kinect est un module fixe, compilé avec le logiciel principal. Dans le script de création de la WFST, il apparaît toujours dans la liste des modules à associer aux commandes vocales, contrairement aux modules python. Le nom "modulekinect" lui est associé dans le fichier de *matching* entre les commandes et les modules. Lorsque celui-ci reçoit la chaîne de caractères "modulekinect" via le *socket*, il émet un signal à la classe de *mapping* afin d'initier la téléopération.

Un autre signal (*typeOfModuleLaunched*), permet à l'interface graphique de savoir en permanence quel type de modules est lancé. Ces types sont représentés par un chiffre :

1. Module Kinect : le module utilise les deux labels pour afficher ses flux vidéo.
2. Module *Face Recognition* : le module utilise un seul label pour son flux vidéo ; l'autre label est rempli avec une image grise unie.
3. Module python : un module python est lancé ; aucun des label n'est utilisé ; les deux labels sont remplis avec des images grises unies.

3.13. Sources de flux audio

Juicer possède différents types d'entrées audio :

- PulseAudio : utilisé par défaut sous Ubuntu, le serveur audio PulseAudio permet d'avoir accès aux périphériques audio d'un PC ; il est utilisé par défaut par Juicer ; de base, c'est donc la source audio sélectionnée par PulseAudio (via le panneau "son" de Ubuntu) qui est active.
- ALSA : couche de flux sonore la plus proche du hardware, puisqu'elle embarque les drivers audio nécessaires à leur utilisation.
- RTAudio : API comportant des classes permettant la manipulation de flux audios.
- SNDFile : cette interface permet de lire un enregistrement audio à partir d'un fichier.

Nous utiliserons par commodité l'entrée principale de PulseAudio, c'est-à-dire le microphone du PC. Un développement futur permettrait d'améliorer ce système de capture et de coupler de nouveaux types de sources au logiciel. Notons que ce développement demanderait l'enregistrement d'une "source audio virtuelle" au sein du système d'exploitation. Cela n'est pas l'objet de ce travail mais permettrait une plus grande flexibilité dans l'utilisation de ce logiciel.

3.14. Script d'exécution

Un script python a été écrit afin de faciliter l'exécution des différents logiciels. Celui-ci utilise le module *subprocess* de python qui permet de lancer un processus à partir du chemin d'un exécutable. L'intérêt de cette méthode réside dans le fait que l'appel de la méthode de création de processus n'est pas bloquante et que nous pouvons, par conséquent, lancer plusieurs applications indépendantes du script lui-même. Lorsque le script est terminé par l'action de l'utilisateur, la fonction *callback* "atExit" permet de terminer tous les processus lancés par le script.

Le script en question lance donc d'abord *Juicer* puis l'application principale décrite dans ce chapitre.

En spécifiant l'argument "demo" au script, celui-ci configure et lance ensuite Naoqi ainsi que le simulateur *choregraph*. Cela permet de lancer un environnement de démonstration fonctionnel en une seule commande.

Nous pouvons voir ici un exemple d'utilisation du script :

```
$ python launch.py demo
```

3.15. Discussion des résultats

Nous allons discuter ici des résultats obtenus avec la commande de Nao par la voix.

Tout d'abord, nous allons discuter des résultats obtenus avec Juicer. Ce n'est pas ici le pourcentage de détection du logiciel qui nous intéresse, puisque son développement

n'est pas le fruit de ce travail. Nous allons plutôt nous intéresser à la stabilité de celui-ci et à la réussite de son intégration au sein du logiciel. Tout d'abord, l'utilisation de *socket* a mis la stabilité de Juicer à rude épreuve (voir plus haut dans ce chapitre). Nous avons cependant pu sécuriser avec succès cette partie du logiciel. Nous avons pu aussi tester cela lors d'un *stress test* improvisé. Le logiciel a reçu un signal audio durant plusieurs heures sans provoquer de *crash* de l'application. Nous considérons donc que le logiciel est stable.

Juicer possède plusieurs types de sources audios (voir plus haut dans ce chapitre). Après étude des différentes possibilités, il s'est avéré difficile de développer une entrée audio générique acceptant tous types de sources dans le temps imparti. Juicer utilise donc exclusivement une source PulseAudio.

4. Face recognition

4.1. Survol

Dans ce chapitre, nous allons discuter de l'intégration d'un algorithme de *face recognition* au sein de l'application de contrôle du robot. Cet algorithme, développé à l'IDIAP, a notamment été utilisé dans le *software KeyLemon* développé par la société du même nom. Il fonctionne sur la base de modèles, créés à partir d'une série d'images d'un visage. Il permet ensuite de reconnaître un visage dans une image en le cherchant dans les modèles connus. Il permet aussi de détecter un visage dans une image et de spécifier la zone dans laquelle ce visage se trouve.

Nous avons dû déterminer un scénario de mise en pratique de cet algorithme avec Nao. La première utilisation possible consiste en la détection de visages contenus dans le champ de vision de Nao et de lui faire tourner la tête dans la direction du premier visage qu'il détecte.

La seconde consiste en "l'apprentissage" des visages qu'il voit afin d'en créer des modèles qu'il pourra reconnaître par la suite.

4.2. Structure

Nous allons ici discuter de la structure de cette partie du logiciel. En effet, celle-ci diffère des autres parties par une utilisation de deux *threads*.

L'utilisation de *threads* permet de ne pas freiner les autres processus lors de l'acquisition de données via les caméras du robot ou lors du traitement de ces données par la librairie *KeyLemon*.

- La classe *Camera* : hérite de *QThread* ; ne fois son *thread* lancé, cette classe s'occupe exclusivement d'acquérir des *frame* provenant de la caméra de Nao.
- La classe *Vision* : hérite de *QThread* ; une fois son *tread* lancé, cette classe récupère les *frames* acquises par la classe *Camera*, les traite et s'occupe de leur passage à travers l'algorithme de *face recognition*.

4.3. Communication avec Nao

Afin de pouvoir acquérir des données provenant des caméras de Nao il nous faut créer un proxy vidéo. A sa création, celui-ci a besoin d'avoir l'adresse et le port du robot.

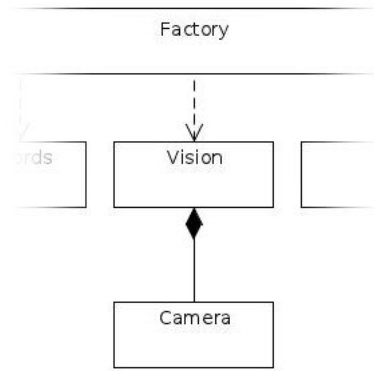


Figure 4.1.: Diagramme de classe partiel. Celui-ci met en évidence la relation entre les *threads*.

Il faut ensuite “l’inscrire” auprès du module vidéo comme un module demandeur de données vidéos. Lors de son inscription, nous devons spécifier un nom correspondant au proxy (celui-ci est utile pour désinscrire le module ensuite), une résolution, un espace colorimétrique ainsi qu’une vitesse d’acquisition.

En précisant ces données, nous pouvons déléguer la transformation de formats d’images au robot ce qui permet de n’envoyer sur le réseau que les données voulues. Par exemple, il serait ennuyeux de devoir envoyer à travers le réseau une image en pleine résolution alors que nous n’avons besoin que d’un petit format.

Nous pouvons ensuite appeler la fonction *getImageRemote* qui va nous retourner la dernière *frame* enregistrée par le robot. Il est possible d’obtenir plusieurs fois la même *frame*, tout dépend évidemment du *frame rate* que nous avons précédemment configuré.

Une fois l’acquisition terminée, il est important de “désinscrire” le proxy auprès du robot car dans le cas contraire, celui-ci conserve les modules considérés comme inscrits jusqu’à son redémarrage.

4.4. Séquencement des actions

L’instance de *Camera* va acquérir des *frames* en provenance des caméras de Nao de façon continue. Chaque *frame* nous arrive du robot sous la forme d’une *ALValue*, une classe propre au *framework* d’Aldebaran qui n’est ni plus ni moins qu’un conteneur de données RGB. Un conteneur de ce type est instancié à la création du thread.

Celui-ci va servir de *buffer* entre le *writer*, c'est-à-dire le proxy vidéo de Nao et le *reader*, c'est-à-dire le *thread* de traitement d'images. A l'arrivée d'une *frame*, nous "fermons" un mutex sur ce *thread* jusqu'à ce que la *frame* ait fini d'être écrite dans son conteneur. Cela empêche que le *reader* appelle la fonction *getFrame* du *thread* et récupère une copie du conteneur dans lequel nous sommes encore en train d'écrire. Dans ce cas, le *reader* est tout simplement mis en attente.

Dans le cas inverse, si le *reader* est en train de lire le conteneur et que la fonction d'acquisition distante est appelée, le *writer* sera mis en attente. Notons que la boucle principale du *writer* est uniquement constituée de l'appel à cette fonction d'acquisition. Nous y avons donc rajouté une temporisation d'une milliseconde afin d'éviter un cas de figure où le mutex serait bloqué en permanence pour le *reader*. A un *frame rate* de 15fps (limitation d'un réseau wifi), soit une période d'environ 66[ms], cela ne freine pas excessivement le processus.

Notons que c'est le *thread Vision* qui envoie périodiquement les *frames* traitées à la vue. Cela permet d'y ajouter un rectangle sur les visages détectés. Cependant, si celui-ci commence un traitement bloquant (p. ex. lors de l'entraînement d'un modèle), il envoie au préalable un signal au *thread Camera* auquel il délègue alors la tâche d'envoyer des *frames* vers la vue.

4.5. Détection de visages

Nous allons aborder ici le fonctionnement de l'algorithme de détection des visages. Celui-ci possède un fonctionnement trivial. Il suffit d'utiliser la fonction *KLFSDK_FaceDetect*. Celle-ci utilise une structure de données propre à cette librairie mais elle utilise le format RGB24 en interne. Il est donc aisé de l'interfacer avec n'importe quel type de sources d'images. Elle retourne un tableau contenant autant d'éléments que le nombre maximal de visages autorisés dans une image. Chaque élément comprend les informations concernant les visages qui ont été détectés ou non, notamment la zone de l'image dans laquelle se trouve le visage. C'est avec ces informations qu'il est possible de dessiner un rectangle autour du visage de l'utilisateur et de contrôler la direction de la tête du robot.

4.6. Reconnaissance faciale

Nous allons nous intéresser dans cette section au fonctionnement des algorithmes de reconnaissance faciale. Nous allons utiliser une méthode de vérification de visages. Celle-ci permet de comparer une *frame* avec un fichier contenant un modèle et de déterminer si ceux-ci correspondent. Cette méthode est en opposition avec la méthode d'identification qui détermine lequel des modèles d'un ensemble est le plus proche d'une *frame*.

Nous avons choisi cette méthode car celle-ci possède un niveau de sécurité variable. En effet, nous pouvons fixer le taux de ressemblance nécessaire à une détection entre une *frame* et un modèle. L'utilisateur va certainement préférer un taux de ressemblance moyen lors de ses essais alors qu'un taux plus bas garantira de meilleurs résultats lors d'une démonstration.

La deuxième raison pour laquelle cette méthode a été choisie réside dans sa gestion des modèles. En effet, alors que la méthode d'identification utilise une base de données, la méthode de vérification utilise directement les fichiers “*.model”. Nous avons donc un stock unique de modèles et celui-ci est toujours à jour.

4.7. Gestion des modèles et affichage des informations utilisateur

Nous allons discuter ici de la gestion de la base de données de modèles. Nous pouvons observer ci-dessous le panneau des modèles dans la GUI. L'idée de ce panneau est de pouvoir facilement visualiser les modèles existants. Chaque fois qu'un visage est “entraîné”, il est enregistré dans un fichier “*.model”. A ce moment la, une *frame* est capturée. Puis un nom d'utilisateur générique lui est attribué (p.ex : “utilisateur 1”). Cette image et ce nom sont ensuite stockés dans ce panneau. Nous faisons correspondre le modèle avec l'élément dans le panneau par leurs index respectifs.

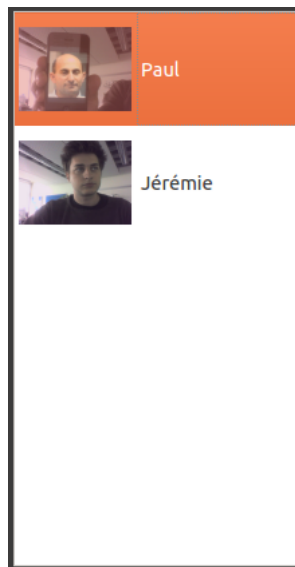


Figure 4.2.: Affichages des modèles existants dans la GUI

Ce panneau est en fait un *QListWidget* dans lequel nous pouvons stocker des *QListWidgetItem* qui ne sont ni plus ni moins que des éléments comprenant une image et une chaîne de caractères, ce qui est parfaitement adapté à notre situation.

Nous pouvons ensuite traiter les *frames* en provenance des caméras du robot afin d'y ajouter des informations sur l'utilisateur. Nous allons tout d'abord afficher un rectangle autour du premier utilisateur que nous détectons. Ce rectangle sera noir si l'utilisateur n'est pas reconnu et blanc dans le cas où l'utilisateur figure déjà dans un des modèles enregistrés.

Si l'utilisateur est correctement identifié, nous pouvons ensuite écrire son nom sur les *frames* dans lesquelles il figure. Ce nom est par défaut le nom constitué à partir de l'index du modèle auquel il correspond. Cependant, l'utilisateur peut facilement modifier ce nom en l'éditant dans le panneau de la GUI.

Nous pouvons observer ci-dessous l'affichage d'une frame dans le cas où l'utilisateur correspond à un modèle enregistré.

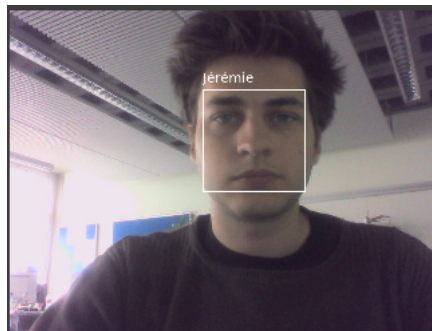


Figure 4.3.: Affichage d'une frame dans laquelle un utilisateur correspond à un modèle enregistré.

4.8. Utilisation de Nao

Nous allons traiter ici de l'interaction entre le module *face recognition* et Nao et des différents types de comportements que nous avons imaginés.

Il existe ici deux types de comportements de Nao. Le premier concerne la direction de sa tête. L'algorithme permettant de détecter les visages sur une image est utilisé ici pour repérer la position de l'utilisateur par rapport au robot.

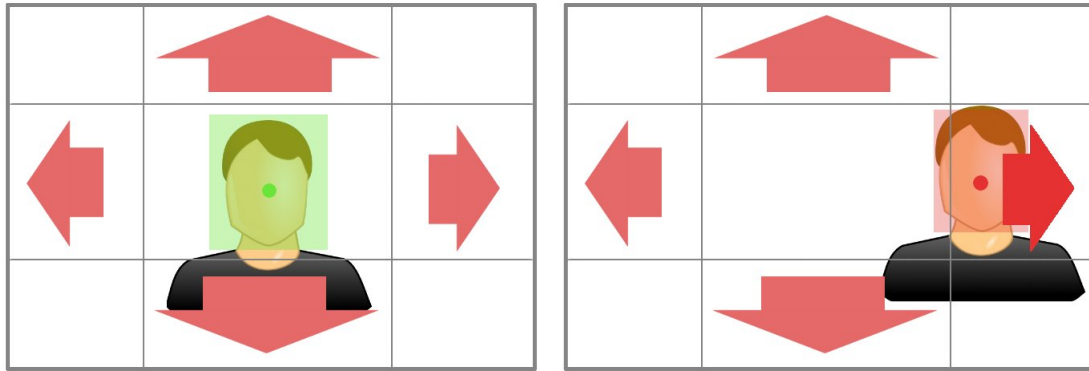


Figure 4.4.: A gauche, une représentation d'un utilisateur centré par rapport à la vue de Nao. A droite, l'utilisateur est à l'extérieur de la zone centrale de la vue de Nao. Dans ce cas là, le mouvement de la tête à droite est amorcé.

Notons tout d'abord que l'utilisateur est représenté par une zone dans les informations qui nous sont retournées par l'algorithme de détection. Nous travaillerons ici uniquement avec un point, c'est-à-dire avec le point central de cette zone. Nous définissons ensuite une zone centrale dans laquelle l'utilisateur est considéré comme centré par rapport au champ de vision de Nao. Si le point symbolisant la position de l'utilisateur sort de cette zone, nous allons amorcer un mouvement dans la direction permettant de recentrer celui-ci. Nous allons donner une consigne de mouvement au robot jusqu'à ce que le l'utilisateur soit retourné dans la zone centrale. Notons que nous lui transmettons un mouvement relatif. Nous allons tout d'abord chercher la valeur de l'angle de la tête de Nao, nous y ajoutons un certain angle, puis nous envoyons la consigne avec cet angle.

Le second comportement que nous définissons pour Nao concerne la reconnaissance des visages. Nous distinguons deux modes de fonctionnement pour ce comportement :

- L'apprentissage automatique des visages qui entrent dans le champ de vision.
- L'apprentissage des visages uniquement lorsque l'utilisateur en donne l'ordre.

Pour illustrer le mode d'apprentissage automatique des visages, observons le diagramme ci-dessous.

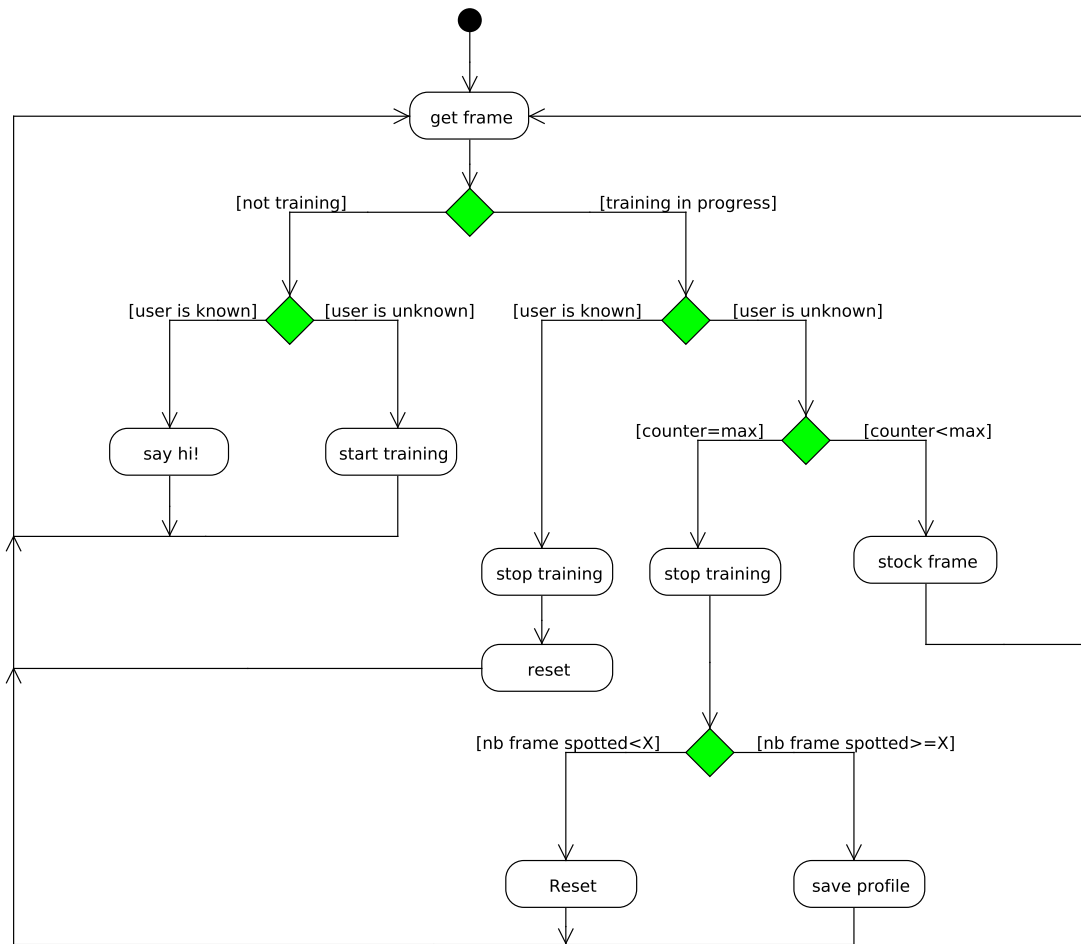


Figure 4.5.: Diagramme d'état représentant l'apprentissage automatique des visages.

Observons le cas le plus trivial. Lorsqu'un utilisateur inconnu apparaît dans le champ de vision, nous commençons un processus d'entraînement de modèle (*training* sur la figure). Ce processus va se dérouler sur un nombre de *frame* défini. Si, durant le processus, un utilisateur est reconnu, le processus s'arrête. Si le nombre de *frame* défini pour l'entraînement est atteint, le processus s'arrête.

Si le processus s'est correctement terminé, il faut ensuite déterminer si, pour le nombre de *frame* acquises, un nombre suffisant d'entre elles comportent un visage correctement détecté par l'algorithme. Si c'est le cas, les *frame* acquises sont considérées comme fiables et un modèle est généré puis enregistré. Si ce n'est pas le cas, les *frame* sont supprimées. Le nombre de *frame* qui doivent comporter un visage détecté est prédéfini. Il est représenté sur la figure par le symbole X.

Les deux paramètres à définir sont donc :

- Nombre de *frame* totales pour un entraînement.
- Nombre de *frame* comportant un visage correctement détecté qui sont nécessaires à un entraînement.

4.9. Formats d'images

Nous allons discuter dans cette partie des différents formats d'images utilisés. En effet, nous travaillerons avec plusieurs formats d'images ce qui implique évidemment la nécessité de pouvoir les convertir. Les formats utilisés sont les suivants :

- QImage : format propre au *framework* Qt, principalement utilisé pour l'affichage dans l'UI.
- IplImage : format d'OpenCV, utilisé lors de l'utilisation de la webcam (mode *debug*).
- AlValue : format propre au *framework* d'Aldebaran.
- raw RGB24 : *data* brutes, représentées par un pointeur (*unsigned char**).

Nous n'allons pas détailler ici toutes les conversions possibles entre ces formats, d'autant que celle-ci sont relativement similaires entre elles. Nous allons cependant étudier un exemple de conversion entre un pointeur *data* brutes de type RGB24 et une QImage.

```
QImage qi(xRes, yRes, QImage::Format_RGB888);  
  
for(int y=0; y<yRes; y++)  
{  
    uchar *imageptr = qi.scanLine(y);  
  
    for(int x=0; x<xRes; x++)  
    {  
        imageptr[0] = ptr[0];  
        imageptr[1] = ptr[1];  
        imageptr[2] = ptr[2];  
  
        imageptr+=3;  
        ptr+=3;  
    }  
}
```

Figure 4.6.: Code source : Conversion d'un buffer RGB en QImage.

Ci-dessus nous pouvons voir ladite conversion. Nous commençons par créer une QImage vide de la bonne taille et du bon format. Nous disposons à ce stade d'un pointeur sur les *data* à copier. Ce pointeur est un *unsigned char**. Cela s'explique par le fait que les données RGB sont construites comme suit.



Figure 4.7.: Construction d'un buffer d'image RGB24

En effet, un char dans ce contexte équivaut à 8 bits de données. En sachant qu'il nous faut 3 couleurs codées sur 8 bits (d'où son appellation RGB24, soit 3×8), le char semble alors être le conteneur idéal pour des données de ce type.

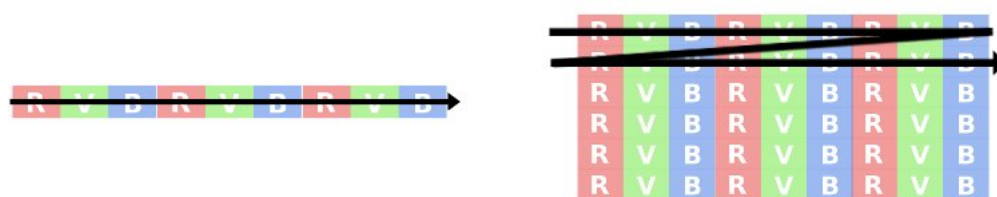


Figure 4.8.: A gauche, la représentation d'un *buffer* RGB en train d'être parcouru. A droite, le même *buffer* affiché dans une image.

Dans tout type de stockage d'une image, les *data* sont "sérialisées". En d'autres termes, elle sont stockées les unes après les autres dans le *buffer*, indépendamment de leur position sur l'axe Y dans l'image. C'est en sachant cela que nous pouvons appréhender le code présenté plus haut. Les deux boucles *for* servent à parcourir l'image sous sa forme affichée (figure de droite) pour stocker les informations de chaque pixel dans un *buffer* de façon sérialisée (figure de gauche). Notons qu'après chaque pixel copié, il est nécessaire d'incrémenter le pointeur de 3 unités (pour les 3 couleurs) afin de passer au pixel suivant.

4.10. Discussion des résultats

Nous allons ici discuter de la performance des algorithmes utilisés ainsi que de leurs limitations. Pour bien comprendre la suite, nous devons tout d'abord préciser certains principes relatifs à ces algorithmes. Tout d'abord, sur une image, un visage ne sera pas détecté si sa taille sur celle-ci est inférieure à 19 pixels. Ensuite, il fait savoir que l'algorithme de reconnaissance faciale travaille exclusivement avec des tailles de visages de 68x68 pixels. Si la taille d'un visage sur l'images est inférieure à cela, la portion d'image comprenant le visage sera "étirée", ce qui ne donne la plupart du temps que des résultats catastrophiques en *face detection*. L'algorithme nous laisse ensuite fixer manuellement la taille maximale d'un visage autorisée. Nous

fixerons donc cette taille proche de la taille verticale de notre image. En sachant tout cela, nous pouvons distinguer quatre cas que voici :

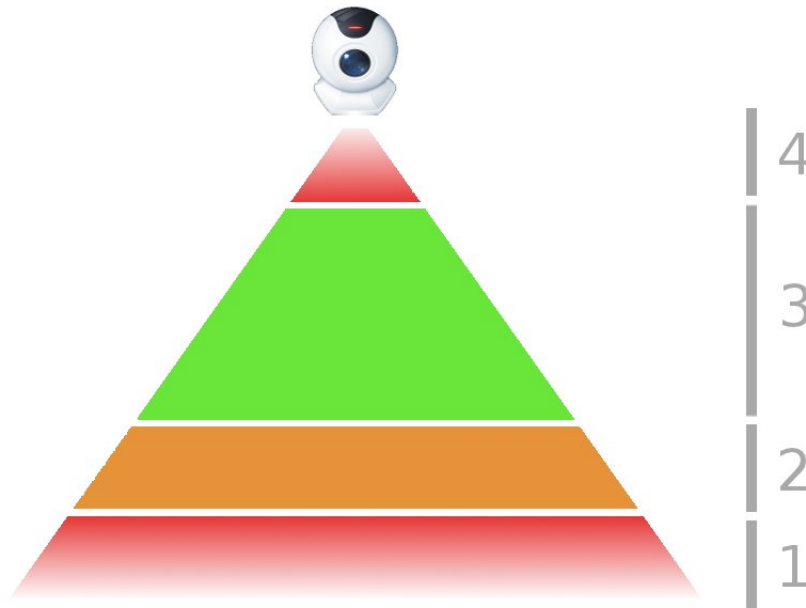


Figure 4.9.: Représentation des distances de détection du visage par une caméra.

1. Le visage fait moins que 19 pixels sur l'image. Celui-ci n'est pas détecté.
2. Le visage a une taille sur l'image comprise entre 19 et 68 pixels sur l'image. Celui-ci est détecté mais la reconnaissance faciale a une chance de réussite très amoindrie.
3. Le visage a une taille sur l'image comprise entre 68 pixels et la taille maximale autorisée. Les résultats sont optimaux en détection et en reconnaissance faciale. Ils le sont d'autant plus en se rapprochant de la taille maximale.
4. Le visage a une taille sur l'image supérieure à la taille maximale autorisée. Celui-ci n'est pas détecté.

Nous allons maintenant faire part de nos observations du taux de détection en fonction de la distance. Notons que le passage en dessous de 68 pixels de la taille du visage produit une chute très rapide des chances de détection. Il est donc assez facile de se faire une idée de la distance minimale de détection.

A une résolution de 320x240 pixels, la distance minimale estimée est de 30 à 35 cm entre la caméra et le visage. Notons que le champ de vision est de 34° sur cette caméra.

En doublant la résolution à 640x480 pixels, la distance minimale de détection double elle aussi et avoisine les 70cm.

Abordons maintenant un problème différent qui affecte le bon fonctionnement de la partie *face recognition* du logiciel. Celui-ci provient du débit du flux vidéo proposé par Nao. Celui-ci, très satisfaisant sur un réseau *ethernet*, chute drastiquement lors d'une connexion à un réseau *wireless*. Nous travaillons sur le réseau *ethernet* à un *frame rate* d'environ 15 fps par seconde, alors que sur un réseau *wireless*, le *frame rate* varie entre 0.5 et 2 fps. Cette estimation a été faite à une résolution de 320x240. Cette limitation est très loin des 11 fps annoncés à cette résolution et en *wireless*. Nous avons effectué un test en ne laissant que le code minimum nécessaire à la capture d'images distantes pour prouver que cette limitation provient bien de l'environnement (bibliothèque d'Aldebaran, couche réseau physique, hardware Nao) et non du logiciel en lui-même. N'ayant pas actuellement trouvé un moyen de contourner cette limitation, le logiciel sera donc uniquement fonctionnel lorsque Nao est connecté via *ethernet* lors de l'utilisation du module de *face recognition*.

5. Conclusion

5.1. Discussion des résultats

Nous allons discuter ici des résultats obtenus lors de ce travail de diplôme. Il est tout d'abord à noter que le cahier des charges initial a évolué depuis le début du projet. En effet, en plus de l'ASR, nous voulions initialement comprendre et peut être mettre en œuvre un algorithme de *headpose recognition*. Cependant, le temps imparti semblait trop court pour adapter un travail de cette envergure. Nous nous sommes donc tourné vers un algorithme de *face recognition*, plus simple à implémenter. Cette décision a été prise d'un commun accord entre les différents intervenants.

Malgré cette modification du cahier des charges en cours de projet, nous pouvons considérer que celui-ci a été respecté. Les deux algorithmes ont été implémentés avec succès au sein de l'application visant à contrôler Nao. De même, cette application est capable de fournir à l'Idiap une méthode de démonstration modulaire et flexible.

Nous exposerons maintenant les problèmes rencontrés et les améliorations possibles. Tout d'abord, dans la partie concernant l'ASR, nous utilisons une source audio du logiciel *PulseAudio*. Ce logiciel, intégré à Ubuntu, nous permet d'utiliser de façon transparente le microphone d'un ordinateur mais ne nous permet pas une grande flexibilité en terme de sources. Nous ne sommes pas en mesure, à l'heure actuelle, de *streamer* un flux audio depuis le robot pour "l'injecter" comme source sonore virtuelle dans le système ou dans le logiciel. Il en est de même pour le *streaming* audio depuis la Kinect. Ceci est donc une amélioration possible pour ce projet.

Pour la partie *KeyLemon*, nous pouvons citer comme problème rencontré le débit du flux vidéo émanant du robot. En effet, le débit étant plus que satisfaisant sur un réseau *ethernet* (~15 fps @ 640x480), celui-ci chute drastiquement sur un réseau wifi (~0.5 - 1 fps @ 320x240). Apparemment, le robot bride le débit en provenance de son interface *wireless* pour pouvoir garantir un flux adéquat à ses 8 *slots* vidéo. Cette limitation impose l'utilisation du robot via un câble *ethernet* lors des démonstrations faisant intervenir la *face recognition*.

5.2. Conclusion

A la lumière de ces résultats, nous pouvons conclure sur une note positive. En effet, nous avons pu mettre en œuvre un démonstrateur fonctionnel dans le délai

imparti et nous avons pu y intégrer un algorithme développé à l'IDIAP. Il sera désormais possible aux visiteurs de l'IDIAP de se faire une meilleure idée des activités de l'institut et de se représenter les domaines étudiés par les chercheurs. C'est aussi dans ce but qu'a été conçu le robot Nao et nous pouvons dire à présent qu'il répond parfaitement aux attentes que peut avoir de lui le milieu de la recherche.

D'un point de vue plus personnel, ce travail a été une chance immense pour moi de pouvoir faire un premier pas dans le domaine de la recherche et d'être baigné durant ces quelques semaines dans l'ambiance très stimulante de l'institut. J'ai pu, dans ce cadre, découvrir de nombreux projets en cours et côtoyer de nombreuses personnes passionnées par leurs travaux. En outre, j'ai aussi pu trouver dans cet environnement toute l'aide nécessaire au bon déroulement de mon projet.

En conclusion, ce travail de diplôme s'est déroulé sans encombre. Nous pouvons nous rendre compte que le délai a été évalué correctement et que l'ajustement du cahier des charges en cours de route a été bénéfique au projet.

Bibliographie

- [1] Nao sdk documentation. [http ://www.aldebaran-robotics.com/documentation/](http://www.aldebaran-robotics.com/documentation/).
- [2] Les fonctions de l'api socket. www.commentcamarche.net/contents/sockets/sockfonc.php3, Mis en ligne le 14 octobre 2008.
- [3] Ros 3d entries : Nao teleop control. www.ros.org/news/2011/01/ros-3d-entries-nao-teleop-control.html, Mis en ligne le 25 janvier 2011.
- [4] J. Koenemann and M. Bennewitz. Whole-body imitation of human motions with a nao humanoid. In *Video Abstract Proc. of the ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2012.
- [5] Magimai.-Doss Mathew Vepa Jithendra Cheng Octavian Hain Thomas Moore Darren, Dines John. Juicer : A weighted finite-state transducer speech decoder, 2006.

6. Annexes

- A** Séquence de configuration de *Juicer*
- B** Détail de l'interface graphique
- C** Guide: Configuration de la démonstration
- D** Guide: Mise en place de la démonstration
- E** Liste des modules python déjà disponibles
- F** Cahier des charges de l'Idiap

A. Séquence de configuration de Juicer

Nous allons ici faire le point sur les différentes étapes de la configuration de Juicer.

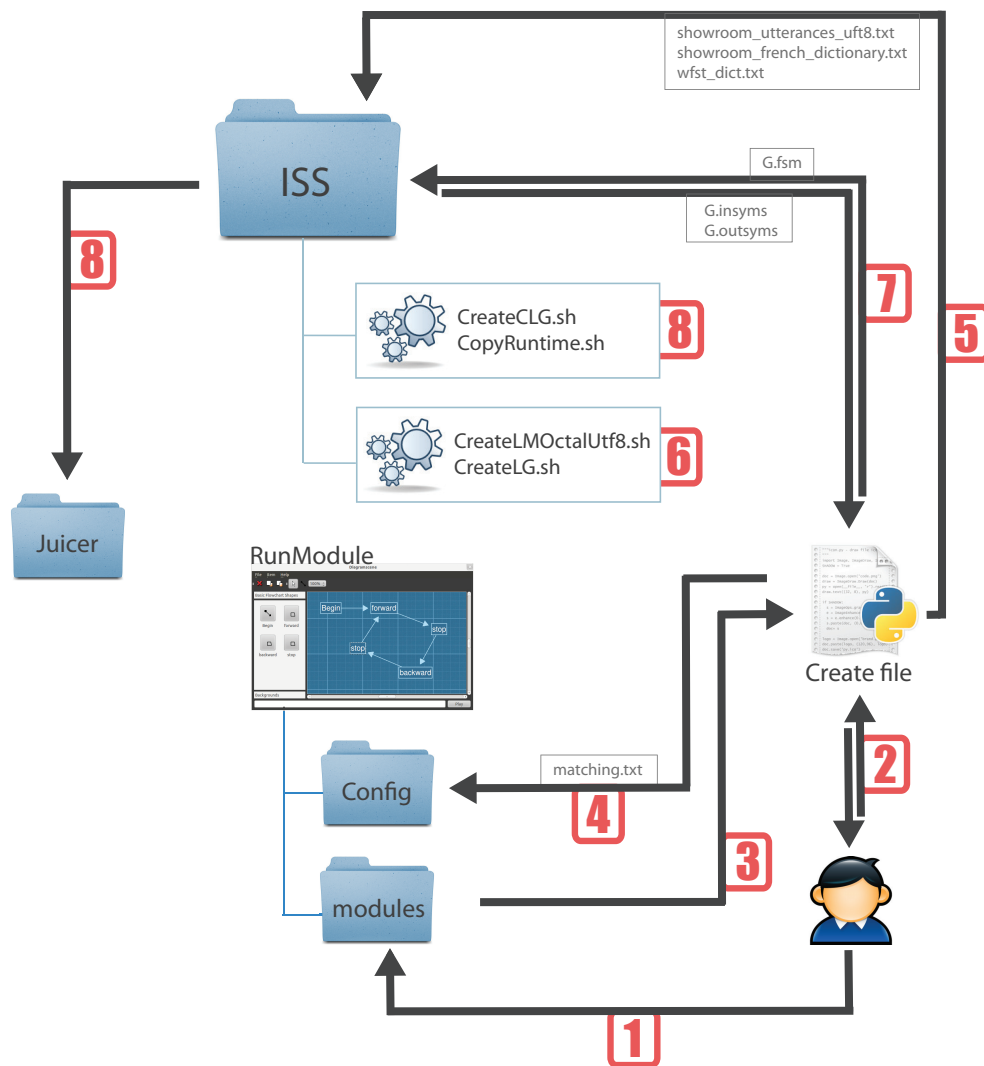


Figure A.1.: Séquence de mise en place des différents fichiers de configuration de Juicer.

La figure ci-dessus représente la chronologie de la configuration de Juicer. Vous

trouverez ci-dessous le descriptif des différentes étapes de cette configuration. Les index de ces étapes font références au index sur la figure. Notons que le descriptif détaillé de la construction de la WFST par le script ne fait pas l’objet de cette section.

1. Étape préliminaire à la configuration. L’utilisateur écrit les modules python nécessaires à sa démonstration. Ceux-ci sont placés dans le dossier “modules” de l’application principale.
2. L’utilisateur lance le script python “create file”.
3. Le script va chercher les modules python disponibles dans le dossier “modules” de l’application principale. Il demande ensuite à l’utilisateur de rentrer les commandes vocales désirées et d’associer chacune de ces commandes à un module.
4. Le script écrit le fichier “matching.txt” qui fait correspondre les commandes vocales avec les modules. Il place ensuite ce fichier dans le dossier “config” de l’application principale.
5. Le script écrit les fichiers “showroom__utterances__utf8.txt”, “showroom__french__dictionary.t” et “wfst__dict.txt” dans le dossier contenant les scripts de Juicer (*ISS* sur la figure).
6. Le script exécute les scripts “CreateLMOctalUtf8.sh” ainsi que “CreateLG.sh”. Ceux-ci permettent de générer les fichiers G.insyms et G.outsyms.
7. En faisant référence au fichier G.insyms et G.outsyms, le script écrit le fichier G.fsm dans le dossier des scripts de Juicer (*ISS* sur la figure).
8. Le script exécute les scripts “CreateCLG.sh” ainsi que “CopyRuntime.sh”. Ceux-ci génèrent le modèle final de la WFST et le place dans le dossier de l’application Juicer.

B. Détail de l'interface graphique

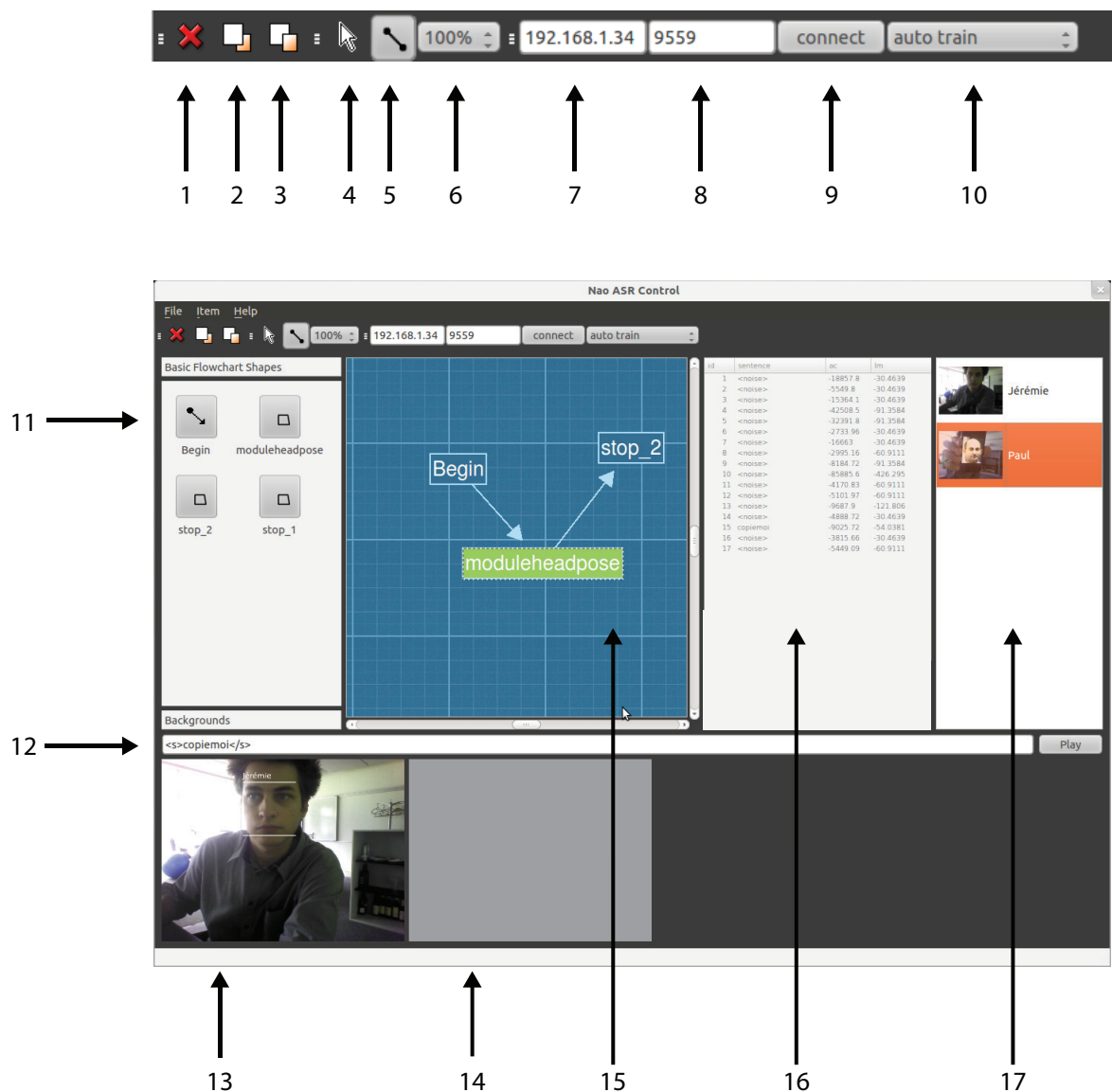


Figure B.1.: Détail de l'interface graphique principale.

1. Supprimer un objet dans la vue.
2. Faire passer un objet à l'avant-plan.

3. Faire passer un objet à l'arrière-plan.
4. Outil de déplacement.
5. Outil de dessin de flèches.
6. Échelle de la vue.
7. IP de Nao.l'IP
8. Port de Naoqi.
9. Bouton *connect* permettant d'envoyer l'IP et le port entré aux différents modules.
10. Sélection du mode d'apprentissage des visages. *Auto train* permet l'apprentissage automatique des visages. *Train by command* permet l'apprentissage uniquement par un ordre.
11. Modules disponibles.
12. Permet d'envoyer des commandes à Nao comme le ferait Juicer. Utilisé pour le *debugging*.
13. Vue caméra (kinect ou Nao).
14. Vue profondeur (kinect).
15. Zone de travail.
16. Enregistrements de la base de données.
17. Enregistrements des modèles de visages.

C. Guide : configuration de la démonstration

Nous allons expliquer ici les étapes de la préparation nécessaire avant le démarrage du logiciel. Tout d'abord, il faut bien comprendre que le comportement de Nao va être déterminé par des commandes vocales. Chacune de ces commandes doit être associée à un module python contenant les commandes nécessaires au contrôle du robot. L'utilisateur doit alors écrire ou se procurer ces modules avant sa présentation.

```
1 def turnBack(IP, port):
2     import motion
3     import math
4     from naoqi import ALProxy
5     motion = ALProxy("ALMotion", str(IP), port)
6
7     motion.setWalkArmsEnabled(True, True)
8     motion.setMotionConfig([["ENABLE_FOOT_CONTACT_PROTECTION", True]])
9
10    X = 0.0
11    Y = 0.0
12    Theta = math.pi
13    motion.post.walkTo(X, Y, Theta)
14
15    return 42
16
```

Figure C.1.: Exemple de module python.

Une fois tous les modules présents, l'utilisateur doit les placer dans le dossier */python/-modules* de l'application. Il lui faut ensuite lancer le script *createfile.py* qui permet d'associer les commandes vocales désirées à ces modules. Notons que ce script nécessite la présence d'un lexicon permettant de faire le lien entre les commandes et leurs équivalents phonétiques. Nous pouvons voir ci-dessous un exemple d'utilisation de ce script.

```
Enter a command to spot(type end to finish):
va tout droit
0. stop.py
1. talkMore.py
2. oneFoot.py
3. turnBack.py
4. handsUp.py
5. turnLeft.py
6. backward.py
7. standUp.py
8. turnRight.py
9. talk.py
10. dance.py
11. forward.py
12. module Kinect
13. module Head Pose
14. training order
Choose a module for the command:
11
```

Figure C.2.: Exemple d'utilisation du script *createfile.py*.

Dans cet exemple, l'utilisateur a entré la commande "va tout droit". Le script a ensuite été chercher tous les modules disponibles et a demandé à l'utilisateur d'associer sa commande avec l'un d'eux.

Une fois toutes les commandes entrées avec succès, l'utilisateur peut taper la commande "end" afin de lancer la séquence de configuration.

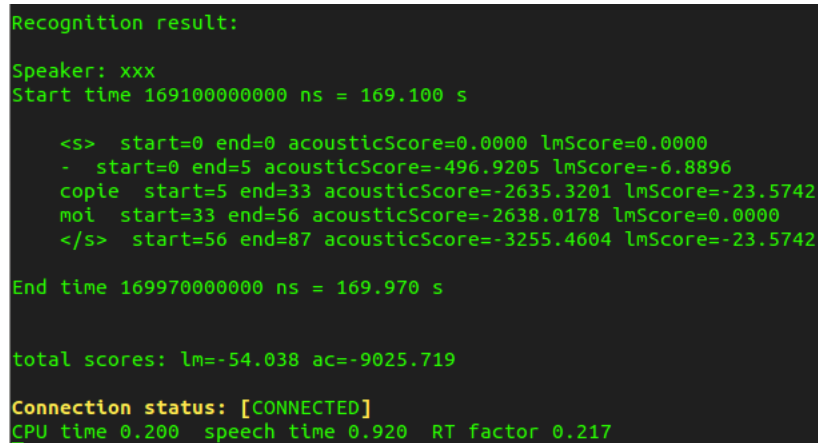
D. Guide : mise en place de la démonstration

Dans cette section, nous allons passer en revue le déroulement d’une démonstration afin d’en comprendre les tenants et les aboutissants.

Il faut tout d’abord lancer le logiciel Juicer par cette commande :

```
$ ./juicer_real_time_cpp.sh
```

Nous pouvons observer ci-dessous Juicer en fonctionnement.



```
Recognition result:
Speaker: xxx
Start time 169100000000 ns = 169.100 s

<s> start=0 end=0 acousticScore=0.0000 lmScore=0.0000
- start=0 end=5 acousticScore=-496.9205 lmScore=-6.8896
copie start=5 end=33 acousticScore=-2635.3201 lmScore=-23.5742
moi start=33 end=56 acousticScore=-2638.0178 lmScore=0.0000
</s> start=56 end=87 acousticScore=-3255.4604 lmScore=-23.5742

End time 169970000000 ns = 169.970 s

total scores: lm=-54.038 ac=-9025.719

Connection status: [CONNECTED]
CPU time 0.200 speech time 0.920 RT factor 0.217
```

Figure D.1.: Juicer en fonctionnement. La commande “copie moi” a été détectée.

Une fois Juicer lancé, il suffit à l’utilisateur de lancer le logiciel principal “RunModule”.

Nous allons maintenant expliquer la création d’un comportement pour Nao lors d’une démonstration. Premièrement, il est indispensable de rentrer une adresse IP et un port permettant l’accès à Nao.

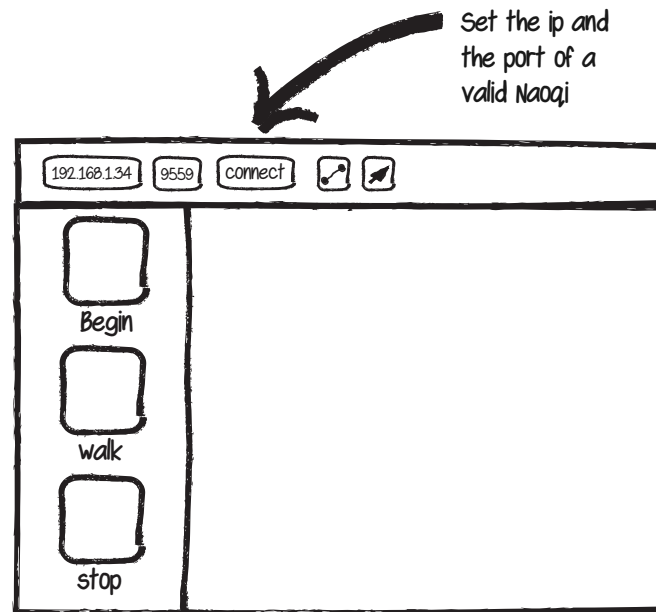


Figure D.2.: Première étape : Entrer l'IP et le port de Naoqi.

Nous pouvons maintenant sélectionner l'outil de déplacement. Grâce à lui, nous pouvons glisser les blocs d'action dans la zone de travail.

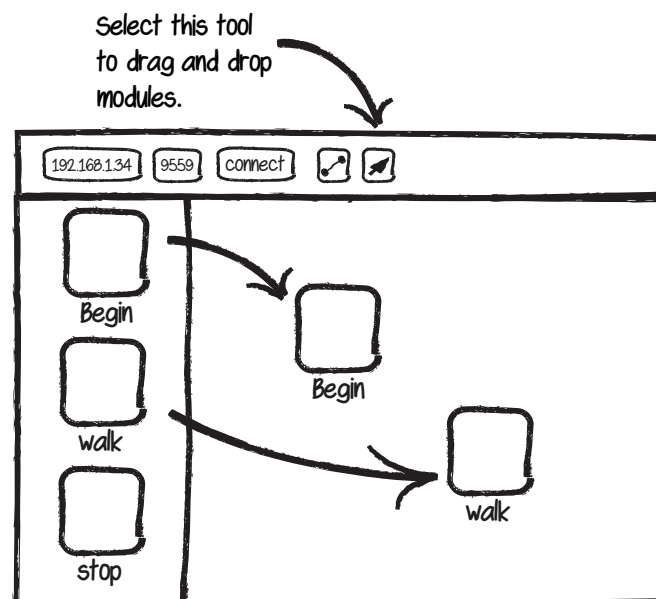


Figure D.3.: Seconde étape : Glisser les modules nécessaires à la démonstration.

Sélectionnons maintenant l'outil de création de flèches. Celui-ci nous permet d'établir un chemin entre les blocs.

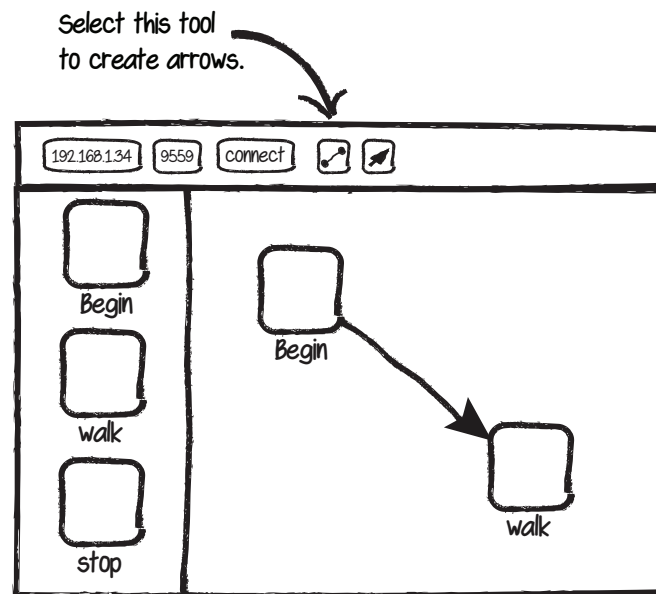


Figure D.4.: Troisième étape : Établissement de la logique de la démonstration en reliant les modules.

A ce stade, le logiciel est prêt à recevoir des commandes vocales. Le logiciel commence toujours à l'état *begin*. Lorsqu'une commande vocale est détectée, le logiciel détermine si celle-ci est associée à un module qui suis l'état actuel. Si c'est le cas, le module est lancé.

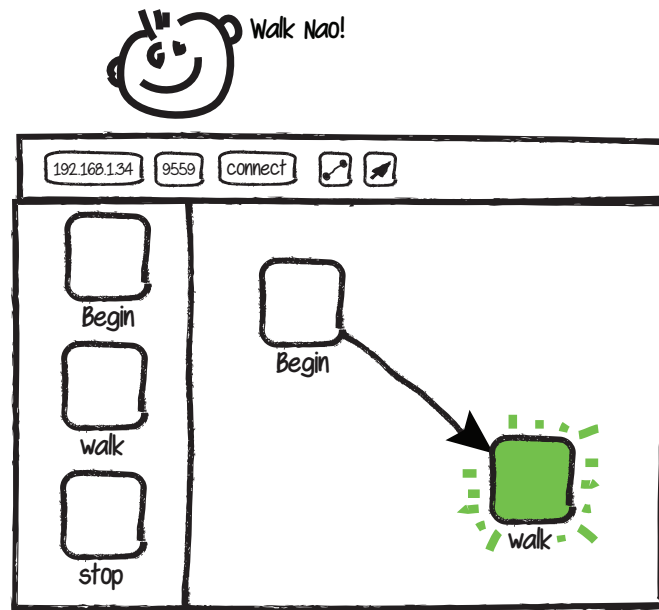


Figure D.5.: Le logiciel est immédiatement fonctionnel. Il suffit alors de prononcer les commandes préalablement définies.

E. Liste des modules python déjà disponibles

Voici une liste des modules qui ont déjà été écrits :

stop : Arrête l'action en cours. Le robot prend ensuite une pose initiale.

forward : Avance tout droit. Doit être stoppé par le modules "stop".

backward : Recule. Doit être stoppé par le modules "stop".

turn_left : Tourne à gauche. Doit être stoppé par le modules "stop".

turn_right : Tourne à droite. Doit être stoppé par le modules "stop".

turn_back : Effectue un demi tour.

hands_up : Lève les bras.

dance : Nao effectue une petite chorégraphie.

kinect : Déploie le module Kinect¹.

face_tracking : Déploie le module de *face tracking/recognition*².

1. Ce module est traité dans le rapport du PrS.

2. Ce module est traité plus loin dans ce document.

Annexe A

Année académique : 2011/12
Etudiant : Jérémie Rappaz
Superviseur HES-SO Valais : Prof. Pierre Roduit
Superviseur Idiap : Flavio Tarsetti
Début du travail à l'Idiap: 14.05.2012
Fin du travail à l'Idiap: 09.07.2012

Titre du travail de diplôme : Développement d'un démonstrateur basé sur le robot Nao

Contenu du travail : L'institut de recherche Idiap (<http://www.idiap.ch>) spécialisé dans les interfaces homme-machine et le traitement de l'information multimédia développe des démonstrateurs technologiques afin notamment de présenter ses recherches au grand public.

Le projet de semestre ayant permis de démontrer que le robot NAO (<http://www.aldebaran-robotics.com>) pouvait facilement être utilisé comme objet de démonstration (reproduction de mouvements humains acquis par une Kinect), l'objectif du projet de diplôme sera de poursuivre avec la réalisation d'un démonstrateur complet basé sur ce robot.

Ce projet de diplôme sera séparé en 2 phases. Lors de la première, l'objectif sera de commander le robot par la voix. Une application, basée sur une librairie existante de l'Idiap, devra être implémentée et permettre de reconnaître les ordres donnés au robot et d'exécuter ceux-ci (e.g. "tourne à droite" devra provoquer une rotation du robot). Il faudra donc lister tous les scénarios (actions réalisables par le robot) et créer une application de commande qui sera capable de passer d'un scénario à l'autre. La stabilité du démonstrateur devra aussi être optimisée, afin que ce démonstrateur soit capable de fonctionner correctement avec plusieurs "interlocuteurs" et durant une durée importante.

Lors de la deuxième phase, qui sera plus exploratoire, un algorithme de « head pose estimation » sera porté sur la plateforme NAO, afin que le robot puisse reconnaître l'orientation de la tête de son interlocuteur et reproduire cette posture.

Les objectifs du travail sont :

- Portage d'un algorithme de commande vocale sur la plateforme NAO
- Implémentation des différents scénarios d'interaction avec le robot et implémentation du système de commande
- Test et analyse de la stabilité du démonstrateur
- Portage d'une librairie de head pose estimation.

Superviseur HES-SO Valais :

Lieu et date : 15.05.2012
My


Signature :



Etudiant :

Lieu et date : 15.05.2012
My

Signature :



Superviseur Idiap :

Lieu et date : Martigny, le 15/05/2012

Signature :

