

EIRBOT
COUPE DE FRANCE DE ROBOTIQUE

Equipe Eirboat

1A 2019-2020



Table des matières

II	Description des projets	2
1	Description générale de l'organisation	3
1.1	Arbres des tâches à réaliser par le robot	3
1.2	Répartition des tâches	5
1.3	Points pour la coupe	7
2	Mécanique	8
2.1	Mécanique générale du robot	8
2.2	Actionneurs	8
3	Electronique	9
3.1	Alimentation	9
3.2	Puissance	10
3.3	Actionneur	10
4	Informatique	11
4.1	Asservissement	11
4.2	Stratégie	11
4.2.1	Modélisation de la table - Définition des obstacles	11
4.2.2	Recherche de Chemin - Algorithme A*	12
4.2.3	Détection des adversaires	13
4.2.4	Gestion des actionneurs	13
4.2.5	Boucle de jeu principale	13
4.2.6	Mise en place de tests	13
4.3	Protocole de communication Stratégie → Asservissement	13
4.3.1	Asservissement	13
4.3.2	Détection	14

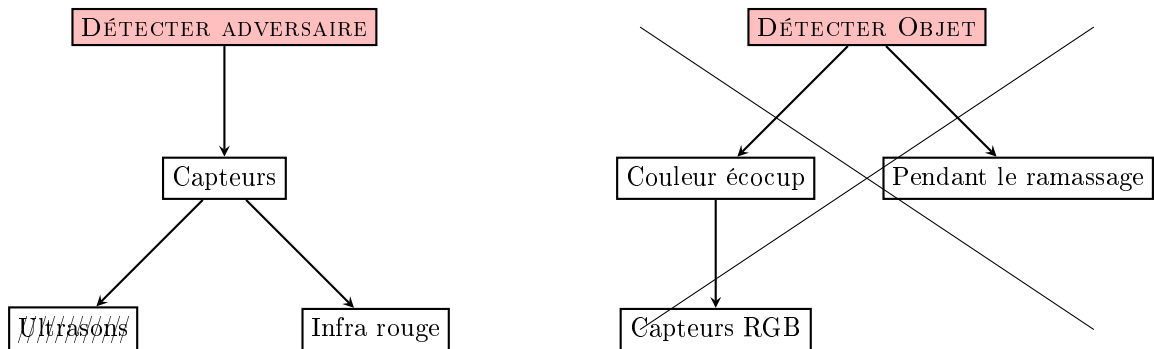
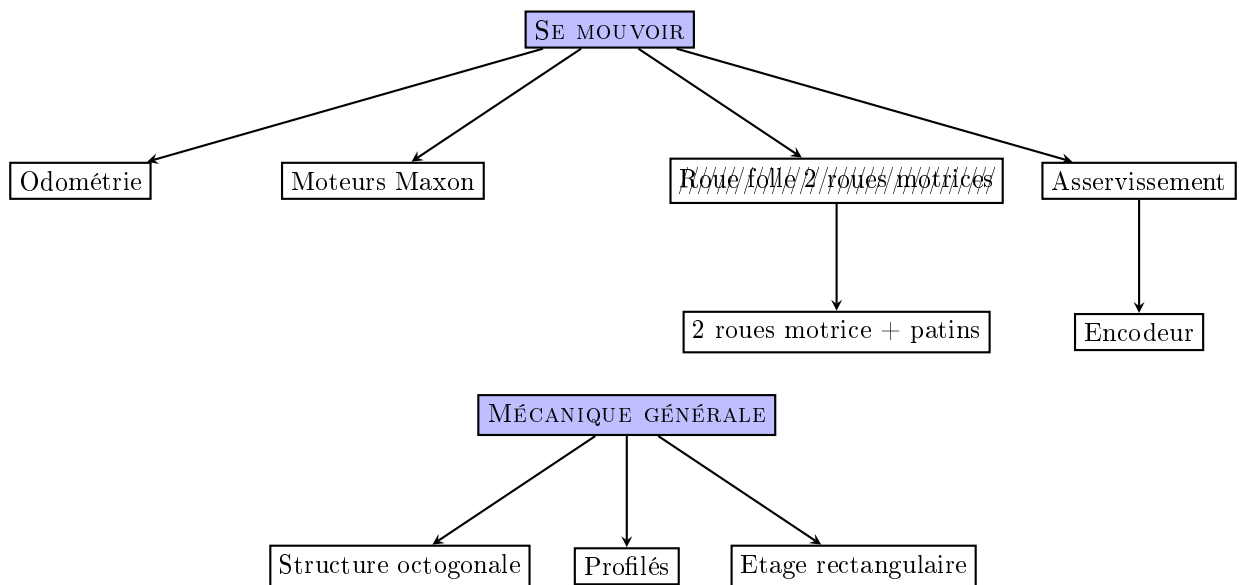
Deuxième partie

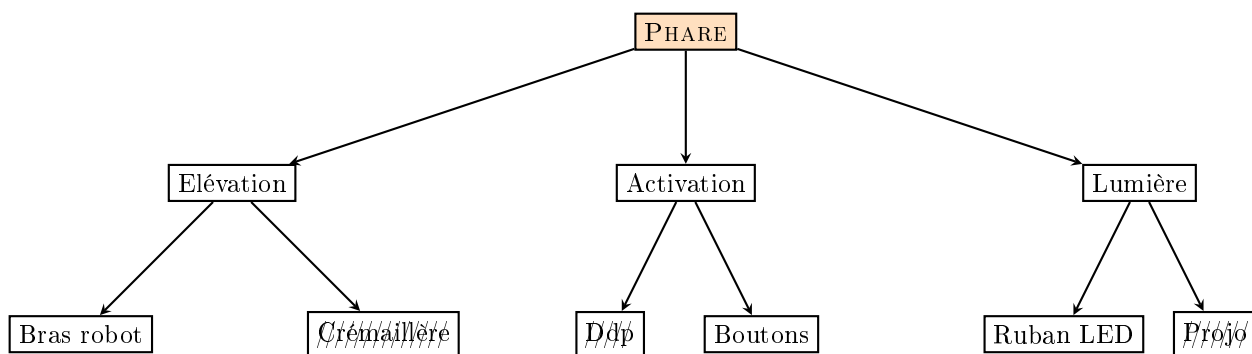
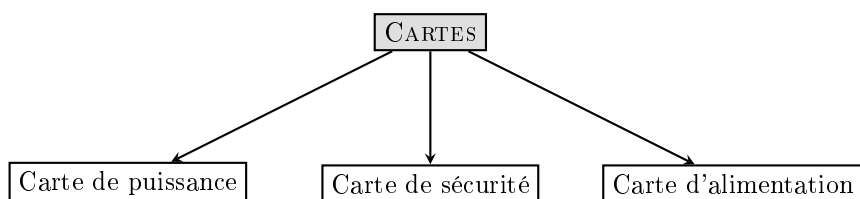
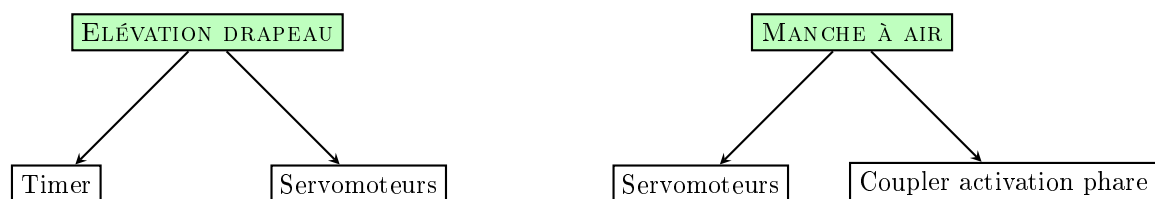
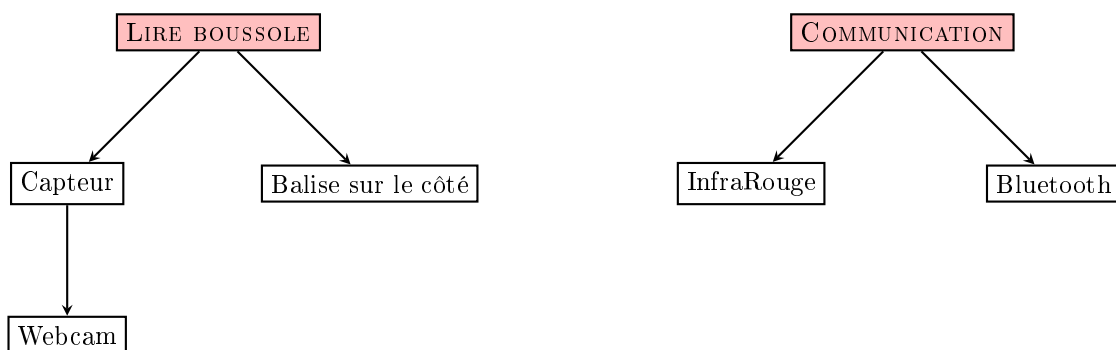
Description des projets

Chapitre 1

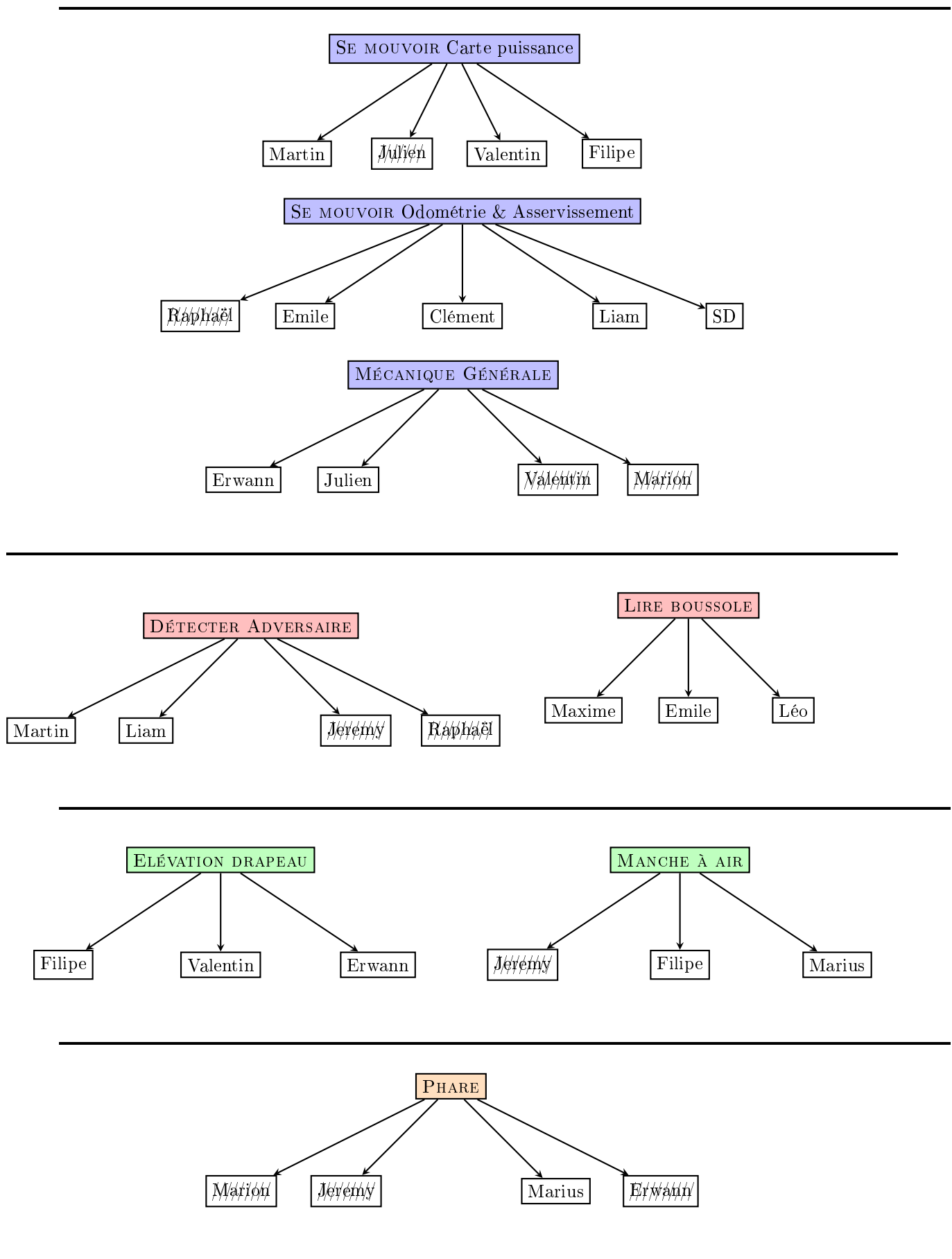
Description générale de l'organisation

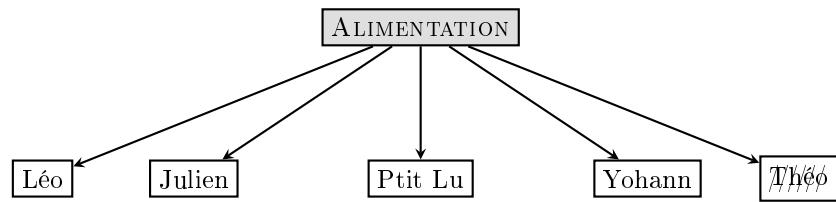
1.1 Arbres des tâches à réaliser par le robot



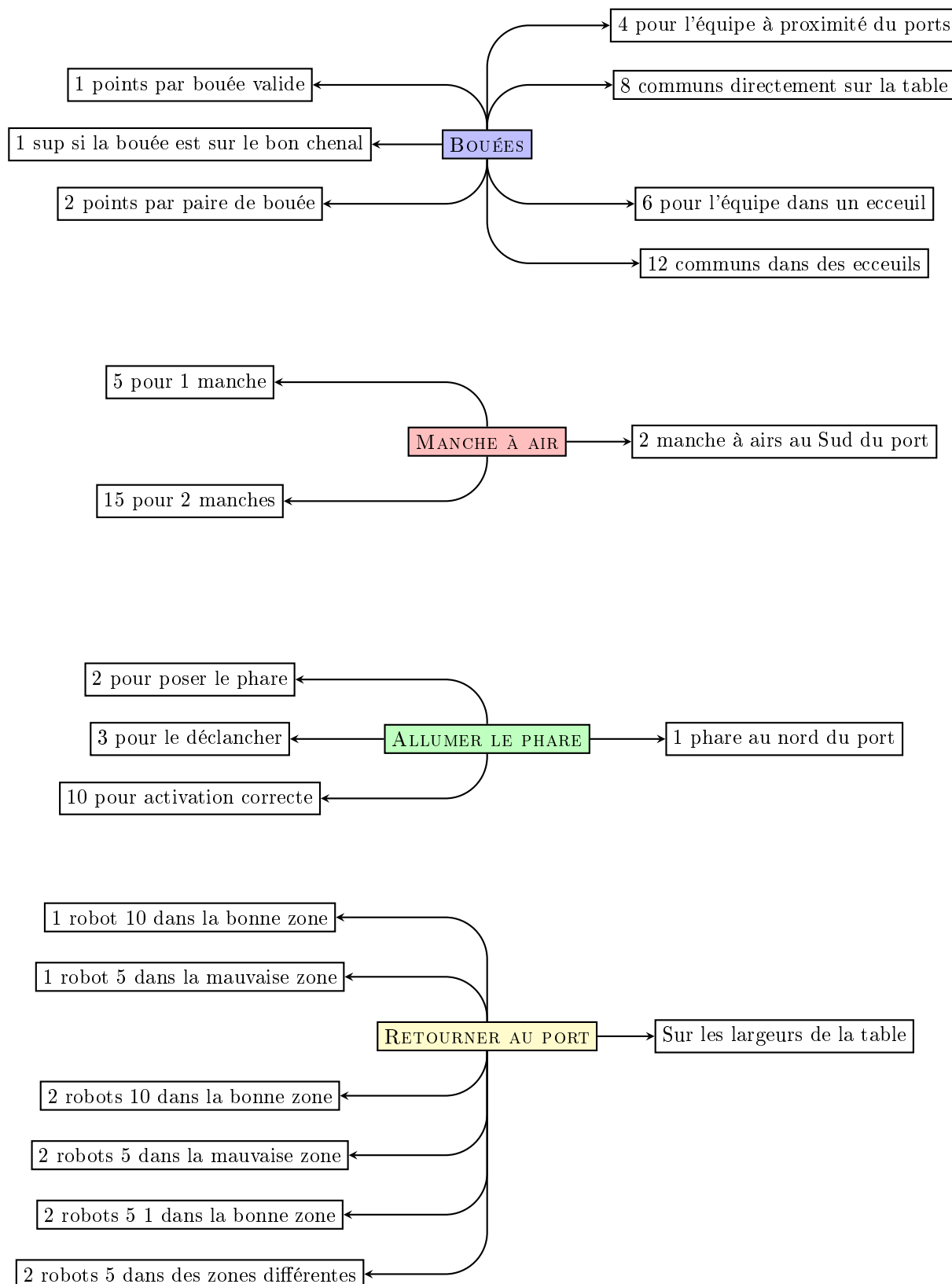


1.2 Répartition des tâches





1.3 Points pour la coupe



Chapitre 2

Mécanique

2.1 Mécanique générale du robot

2.2 Actionneurs

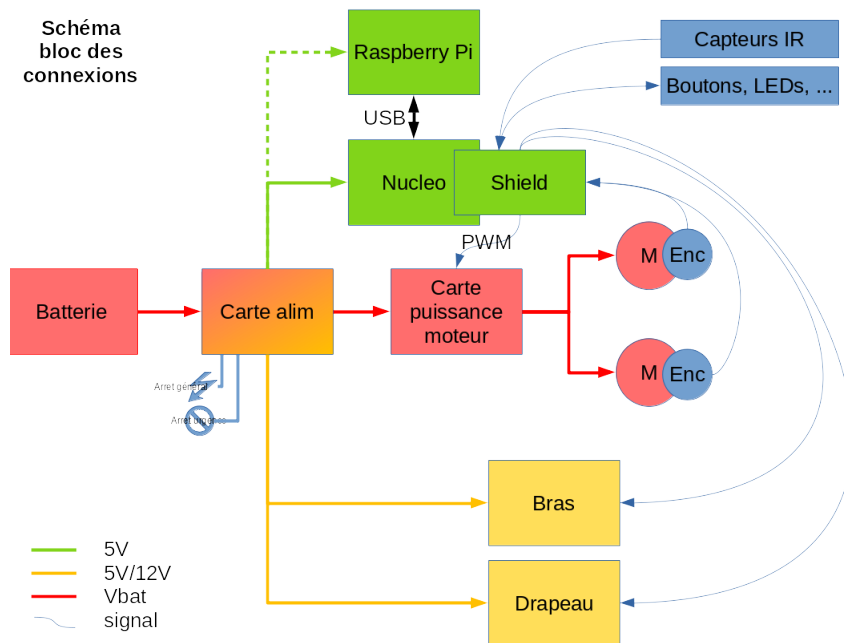
Chapitre 3

Electronique

3.1 Alimentation

Objectif. La carte d'alimentation doit distribuer l'énergie et convertir les tensions. Elle récupère l'énergie de la batterie et elle possède les caractéristiques suivantes :

- Moteurs : 3A/moteurs
- Logique 5V 1.5A
- Puissance 5V 3A
- Puissance 12V 3A



Les détails techniques sont disponibles dans le dossier [alimentation](#) sur github.

3.2 Puissance

3.3 Actionneur

Chapitre 4

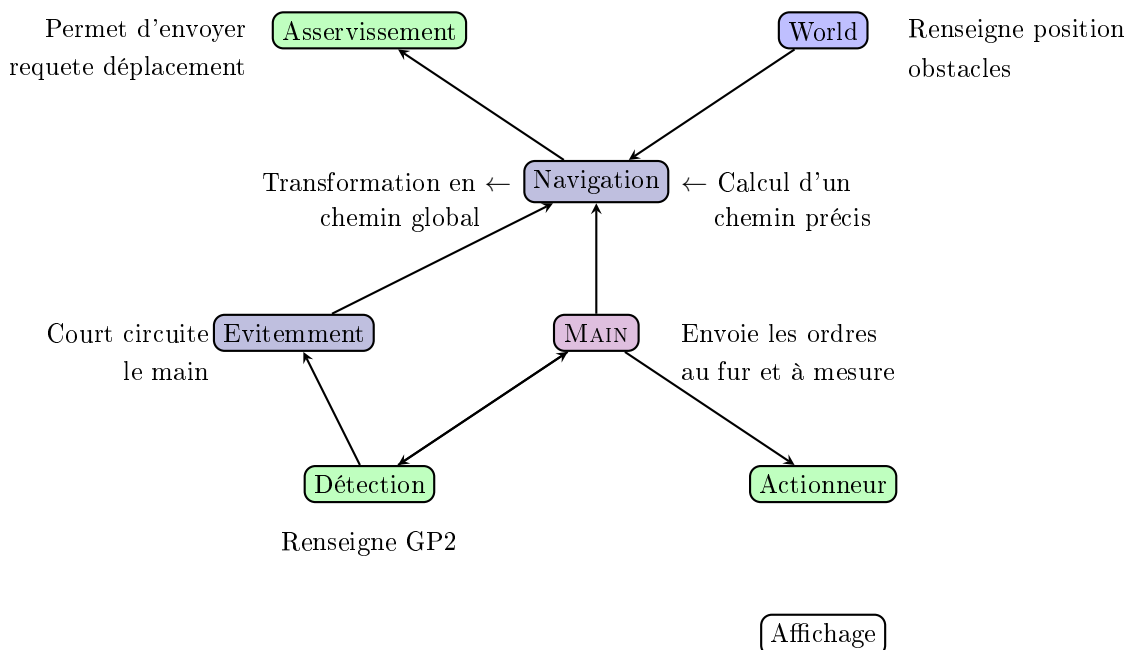
Informatique

4.1 Asservissement

4.2 Stratégie

Introduction

Cette partie à pour objectif de définir de manière abstraite la communication entre les différents blocs informatique pour permettre une compréhension générale par tous (le violet représente le plus haut niveau, le vert représente un point de communication avec la nucléo).



4.2.1 Modélisation de la table - Définition des obstacles

Objectif. Cette section de la stratégie est la première section, elle a pour but de définir l'environnement dans lequel le robot va évoluer. Cette définition de l'environnement sera cruciale pour la recherche de chemin.

L'objectif est donc de créer un système permettant de définir des obstacles et des méthodes permettant de vérifier si il y a des obstacles à un endroit.

Définition d'un obstacle. La structure permettant de définir un obstacle est relativement simple, elle contient la position du centre d'un obstacle, sa largeur et sa longueur. Avec cette définition tous nos obstacles sont rectangulaires. Par la suite, nous avons renseigné les positions de tous les obstacles et leurs types (eco cup, petit taquet, grand taquet). Nous mettons toutes ces informations dans un vecteur. Ce vecteur sera l'élément de base pour détecter les collisions entre notre robot et les obstacles.

Prévision de collision. L'idée est d'obtenir une méthode permettant de savoir si notre robot peut aller à un point donné sans toucher d'obstacles. Nous réalisons un test mathématique simple qui permet de nous donner en fonction de coordonnées x, y , d'une forme d'obstacle et d'une liste contenant tous les obstacles si la case est valide ou non. Le fait de pouvoir tester les différentes formes d'obstacles nous permet dans des cas critiques de désactiver les tests sur les objets qui sont mobiles et ainsi nous sortir d'une situation délicate.

En somme nous disposons maintenant d'une modélisation simple de la table ce qui nous permettra de pouvoir calculer les déplacements du robot tout en évitant les obstacles. Nous ajoutons aussi une fonctionnalité permettant de ne pas prendre en compte toutes les obstacles ce qui nous permet de prioriser l'évitement de certains obstacles (nous préférons rentrer en collision avec une *eco_cup* qu'avec un robot).

4.2.2 Recherche de Chemin - Algorithme A*

Objectif. L'idée est d'implémenter un algorithme permettant de calculer un chemin pour notre robot. L'idée est qu'il puisse, d'un point donné aller à un autre point tout en évitant les obstacles. Ces obstacles seront soit défini directement (comme les gobelets) ou défini en fonction de la détection des robots adverses. Nous réutilisons la classe *world* pour l'implémentation des obstacles.

Description de l'algorithme de pathfinding A*. A*¹ commence à un nœud choisi. Il applique à ce dernier un coût initial, il estime ensuite la distance entre ce nœud et le but à atteindre. Le coût additionné à l'évaluation représentent le *cout euclidien* assignné au chemin menant à ce nœud. Le nœud est alors ajouté à une file d'attente prioritaire, appelée *open list*.

Premièrement l'algorithme récupère le premier nœud de l'*open list*. Si elle est vide, il n'y a aucun chemin du nœud initial à celui d'arrivée, l'algorithme est en erreur. Si le nœud est celui d'arrivée, l'algorithme va reconstruire² le chemin complet et renvoyer le résultat.

Ensuite, si le nœud n'est pas le nœud d'arrivée alors de nouveaux nœuds sont créés pour tous les nœuds contigus admissibles³. L'A* calcule ensuite son coût et le stocke avec le nœud. Ce coût est calculé à partir de la somme du coût de son ancêtre et du coût de l'opération pour atteindre ce nouveau nœud.

1. https://fr.wikipedia.org/wiki/Algorithme_A*

2. Cette reconstruction se fait grâce aux informations de la *closed list*

3. Dans notre cas si il est libre ou non



En parallèle l'algorithme conserve la liste des noeuds qui ont été vérifiés, c'est la *closed list*. Si un noeud nouvellement produit est déjà dans cette liste avec un coût égal ou inférieur, on ne fait rien.

Après, l'évaluation de la distance du nouveau noeud au noeud d'arrivée est ajoutée au coût pour former l'heuristique du noeud. Ce noeud est alors ajouté à la liste d'attente prioritaire, à moins qu'un noeud identique dans cette liste ne possède déjà une heuristique inférieure ou égale.

Une fois ces étapes effectuées pour chaque nouveau noeud contigu, le noeud original pris de la file d'attente prioritaire est ajouté à la liste des noeuds vérifiés. Le prochain noeud est alors retiré de la file d'attente prioritaire et le processus recommence.

Mise en oeuvre. Les détails techniques sont disponibles dans le dossier code, fichier [navigation](#).

Reprennons l'idée générale de cette partie : implémentation d'une recherche de chemin. L'algorithme qui au coeur de cette recherche de chemin à déjà été présenté. Nous allons donc décrire la classe *Navigation*, le but de cette classe est de récupérer une destination souhaitée, de trouver le chemin (tout en évitant les obstacles) via l'A* et enfin d'effectuer les déplacements en appelant la classe d'Asservissement via le protocole de communication. Cette classe aura donc des noeuds comme attribut, (un noeud contient la position x, y , les différents cout $fcost, gcost, hcost$ et les parents du noeud $parentX, parentY$). Elle possède 3 méthodes principales, tout d'abord Astar qui permet d'exécuter l'Astar, cette méthode appelle MakePath qui a pour but de construire un vecteur contenant le chemin final. Cepedant à ce stade nous avons une description du chemin très précise (centimètre par centimètre), nous devons fournir quelque chose de moins précis à l'asservissement (au risque d'avoir un papybot). Pour cela nous avons la méthode NavigateToAsserv qui permet de transformer ce chemin en suite de segment ce qui nous permet de nous déplacer.

A ce stade nous avons un moyen de trouver notre chemin et faire deplacer un robot de manière assez brutale, il faudrait développer un algorithme de lissage de la trajectoire (pour avoir des changements de direction moins brutaux).

4.2.3 Détection des adversaires

4.2.4 Gestion des actionneurs

4.2.5 Boucle de jeu principale

4.2.6 Mise en place de tests

Objectif.

4.3 Protocole de communication Stratégie → Asservissement

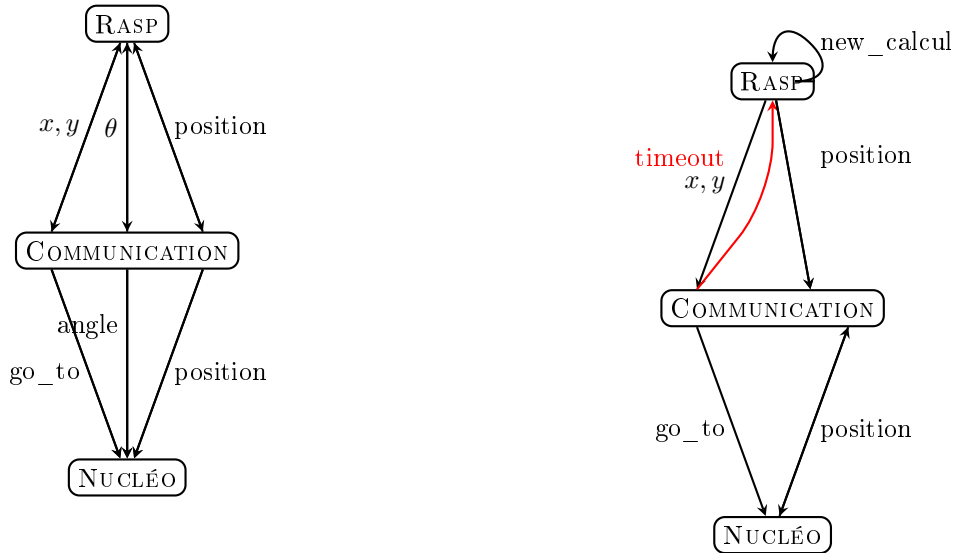
4.3.1 Asservissement

Commençons par le cas d'utilisation classique, nous estimons que la nucléo ne rencontre aucun problème particulier. Dans ce cas normal la rasp peut envoyer trois types de requêtes, une de position, une d'angle et une



d'initialisation. Le protocole de communication transforme ces requêtes en instructions pour la nucléo. La nucléo exécute les actions et renvoie une confirmation d'exécution de la requete ce qui informe la rasp que l'action a correctement été réalisée.

Dans un second temps nous présentons le mécanisme qui se met en place si la rasp transmet un ordre mais qu'aucune réponse n'est receptionnée sur une action.



Il faut encore gérer le cas où la nucléo ne répond plus du tout, nous réalisons alors un branchement, si nous obtenons une succession de timeout alors nous arrêtons la rasp et nous lançons un programme d'urgence pour que la nucléo fasse un programme minimal (c'est à dire rentrer au port). Nous devons encore définir comment activer ce programme minimal.

4.3.2 Détection

Le protocole de communication pour la détection va fonctionner de manière très similaire à l'asservissement. La principale différence réside dans le fait que la nucléo peut envoyer une interruption à la rasp pour lui indiquer que l'on détecte un robot adverse.

Dans le cas d'une détection, automatiquement la nucléo va stopper les moteurs et envoyer à la rasp l'information via le protocole de communication. La rasp va alors recevoir les informations de tous les GP2 actifs sur les différentes distances seuils. Une fois cela effectué, la rasp va déterminer la position et l'angle du robot, elle va alors pouvoir créer un nouvel obstacle et recalculer une trajectoire tout en évitant le robot. Si l'on évite correctement le robot adverse nous continuons notre trajectoire. Dans le cas contraire, les GP2 redetecte quelque chose, la nucléo renvoie une interruption etc. Si l'on arrive dans un cas de figure où nous avons des interruptions sans arrêt alors nous réalisons une manoeuvre d'évitement de grande ampleur. Qui consiste à reculer et faire un détour pour être sûr de l'éviter.

Il reste un dernier cas à traiter si lorsque nous détectons un robot et que la navigation de la rasp ne trouve pas de chemin. Dans ce cas nous devons tout de même nous sortir de la situation. Ainsi la navigation va supprimer les obstacles du type `eco_cup` et s'échapper tout en renversant les écocup.

Ainsi nous avons théoriquement un système qui nous permet de détecter et d'éviter un robot adverse.