Identification, planification et suivi de trajectoire avec un robot.

Lucas ARIES
École d'ingénieur informatique
TELECOM Nancy
Villers-lès-Nancy, France
Email: lucas.aries@telecomnancy.eu

Aurélie DEMURE
École d'ingénieur informatique
TELECOM Nancy
Villers-lès-Nancy, France
Email: aurelie.demure@telecomnancy.eu

Julie ZHEN
École d'ingénieur informatique
TELECOM Nancy
Villers-lès-Nancy, France
Email: julie.zhen@telecomnancy.eu

Amine BOUMAZA
Laboratoire lorrain de recherche
en informatique et ses applications
LORIA, Campus scientifique BP 239
Vandoeuvre-lès-Nancy Cedex, France
Email: amine.boumaza@loria.fr

***Abstract*—L'objectif de cette étude est de développer un système de contrôle autonome pour un robot, en utilisant un programme informatique conjointement avec des données provenant d'une caméra. Ce système vise à permettre au robot de naviguer de manière autonome d'un point A à un point B en utilisant diverses solutions techniques.**
Mots-clés: Robot Thymio, Robotique, Traitement d'image, Planification et suivi de trajectoire

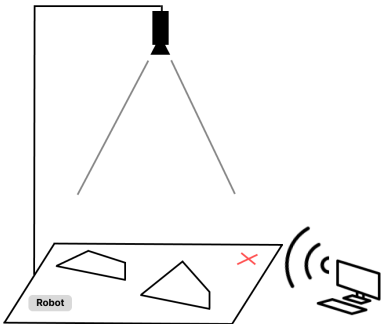

Fig. 1. Représentation de l'arène

I. Introduction

Dans ce projet, nous souhaitons réaliser un prototype de plate-forme pour l'étude de la coordination robotique. Il s'agira d'une arène qui permettrait de faire évoluer un robot Thymio2 dans des environnements modulables. D'un point de vue technique, l'arène sera dotée d'un système de captation par caméras pour localiser les objets et le robot. Un ordinateur se chargera de la programmation et la communication avec le robot.

A. Les problématiques

1) Communication avec le robot: Dans ce contexte, notre objectif est de développer un système de contrôle pour un robot nécessitant une communication permanente avec une source externe, plutôt que de téléverser un programme directement sur le robot pour une autonomie complète. Ce système doit permettre au robot de naviguer de manière autonome d'un point A à un point B en utilisant des informations provenant d'une caméra et en employant diverses solutions techniques.

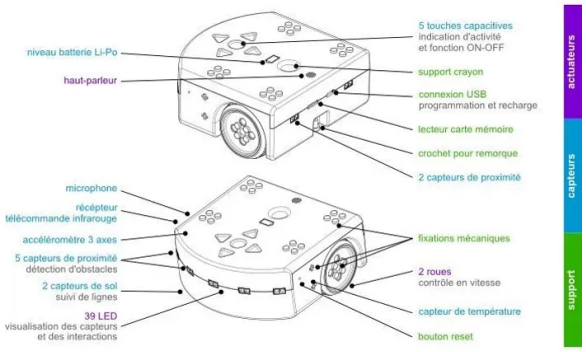

Fig. 2. Représentation d'un robot Thymio2

2) Calibration de la caméra: Ensuite, afin d'avoir une bonne détection de l'environnement du robot, il faut avant tout procéder à la calibration de la caméra. En effet, le contraste entre le repère du monde réel et le repère plan de l'image ainsi que les paramètres internes de la caméra peuvent fausser notre détection.

3) Détection des obstacles et du robot: Il faut explorer la problématique de la détection et de l'identification d'objets dans une scène à l'aide d'une caméra positionnée en hauteur pour obtenir une vue en deux dimensions. Plus spécifiquement le défi de repérer un robot ainsi que divers obstacles dans un environnement donné.

4) Planification de trajectoire: La problématique soulevée ici est la nécessité de planifier la trajectoire d'un robot en utilisant les données fournies par une caméra. Cette planification

requiert la conversion de ces données dans un format approprié pour les algorithmes de planification classiques.

5) Suivi de trajectoire: La problématique abordée concerne l'optimisation de la navigation du robot afin qu'il suive la trajectoire planifiée de manière fluide, sans heurts ni à-coups. L'enjeu réside dans la réalisation d'un contrôle précis et efficace du robot pour garantir une expérience de déplacement homogène et sans accrocs.

II. Communication avec le robot

Dans notre contexte actuel, nous identifions la communication entre le robot et son environnement comme un élément crucial pour le développement d'une solution efficace. Cependant, nous devons prendre en compte des contraintes physiques et logicielles.

La contrainte physique principale est la portée entre le robot et son ordinateur, ainsi que l'utilisation d'une caméra pour la perception de l'environnement. L'utilisation d'un câble pour la communication entre le robot et l'ordinateur est donc exclue, car cette solution limiterait considérablement la mobilité du robot et ajouterait des informations inutiles sur le flux vidéo.

Pour répondre à cette contrainte, nous utilisons une connexion sans fil pour la communication entre le robot et l'ordinateur. Le robot Thymio offre déjà une solution de communication sans fil, ce qui facilite la mise en œuvre de notre solution. Nous utilisons la technologie Bluetooth pour assurer une communication fiable et efficace entre le robot et l'ordinateur.

Dans le cadre de notre projet, nous devons assurer une communication permanente entre le robot et l'ordinateur afin de pouvoir adapter la trajectoire du robot en temps réel en fonction des changements du terrain. En effet, le robot doit être notifié d'un nouveau chemin calculé par l'ordinateur et être en mesure de changer de trajectoire en conséquence.

C'est pourquoi nous ne pouvons pas téléverser un programme au robot et le laisser se déplacer de manière autonome, car cela ne répond pas aux exigences de notre projet. Pour répondre à cette problématique, nous utilisons la librairie ThymioDirect qui permet de communiquer en permanence avec le robot et de lui donner des instructions depuis l'ordinateur en temps réel et sans délai. Cette bibliothèque nous permet de contrôler le robot à distance et de mettre à jour sa trajectoire en fonction des données envoyées par l'ordinateur.

Dans un souci de facilité de développement et de compréhension, nous avons créé une classe pour le robot qui sert d'adaptateur à la bibliothèque. Cela facilite considérablement l'utilisation du robot dans le projet en fournissant une interface simple (il n'est pas nécessaire de configurer le robot et de se reconnecter à chaque fois que l'on souhaite lui donner de nouvelles instructions).

III. Calibration de la caméra et traitement de la distorsion

Dans cette partie, nous allons chercher à calibrer notre caméra, c'est-à-dire trouver ses paramètres internes et externes afin de recouvrir la géométrie des scènes que nous étudierons.

A. Modélisation d'une caméra

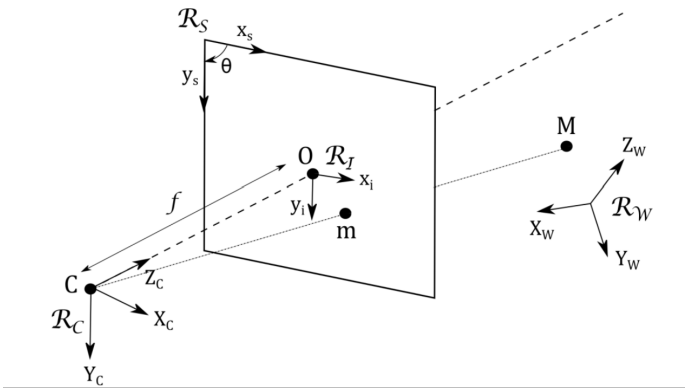

Fig. 3. Modélisation d'une caméra

Légende :

- repère Rc : repère de la caméra avec (Xc, Yc, Zc)
- repère Rw : repère monde "world" qui représente la 3D avec (Xw, Yw, Zw)
- repère Ri : repère de l'image qui représente la 2D avec (xi, yi)
- repère Rs : repère du capteur "skew" avec (xs, ys, theta)

B. Explication de l'origine de la matrice de projection de la caméra

La matrice de projection de la caméra est obtenue grâce à trois transformations. Ces dernières sont détaillées dans le document dédié au traitement d'image donc nous n'aborderons que la première ici.

1) Première transformation: Si nous reprenons la figure, le point M est défini en 3D par le repère "monde" Rw avec ses coordonnées (Xw, Yw, Zw). Cette même position est exprimée dans le repère local Rc de la caméra C par les coordonnées locales (Xc, Yc, Zc). La première transformation est donc ce changement de repère qui donne une matrice T que nous pouvons décomposer en une matrice de rotation R et un vecteur de translation t.

L'obtention de la matrice se fait par le calcul suivant :

$$\begin{Bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{Bmatrix} = [T] \begin{Bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{Bmatrix} = \begin{bmatrix} [R] & \{t\} \\ \{\mathbf{0}_{1\times3}\} & 1 \end{bmatrix} \begin{Bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{Bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{Bmatrix}$$

Fig. 4. Calcul pour la matrice de la première transformation : W -> C

Il y a donc six paramètres : 3 angles et 3 translations qui sont les **paramètres extrinsèques** permettant de définir le positionnement de la caméra dans l'espace 3D.

2) Matrice finale: Après calculs, la matrice de projection M de la caméra est donc la suivante :

$$[M] = \begin{bmatrix} r_{11}f_x + r_{31}c_x & r_{12}f_x + r_{32}c_x & r_{13}f_x + r_{33}c_x & t_x f_x + t_z c_x \\ r_{21}f_y + r_{31}c_y & r_{22}f_y + r_{32}c_y & r_{23}f_y + r_{33}c_y & t_y f_y + t_z c_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix}$$

Fig. 5. Matrice M explicite

C'est donc cette matrice que nous chercherons à obtenir avec différentes librairies et fonctions afin d'étalonner notre caméra.

C. Etude des trois principales méthodes d'étalonnage

1) Etalonnage par mire: L'étalonnage par mire vise à se servir d'un objet de géométrie connue appelé **mire d'étalonnage**. Un damier est très souvent utilisé car il présente des points 3D spécifiques à chaque intersection de lignes verticales et horizontales. Ensuite, un algorithme permet de déterminer les paramètres de la caméra pour lesquels l'écart entre la position des points 3D projetés et la position réelle des points est minimal. Dans les faits, on minimise la somme quadratique des erreurs de reprojection. Le système se complique dans le cas où l'on fonctionne avec deux caméras mais ce n'est pas notre cas.

2) Auto-étalonnage: L'auto-étalonnage vise à utiliser la pièce à tester comme objet d'étalonnage. Il faut pour cela une description dite **dense** de l'objet telle qu'un maillage. On cherche ainsi à trouver simultanément les paramètres implicites de la caméra. Cette fois, on cherche à minimiser l'écart en niveau de gris entre la projection et l'image de référence.

3) Etalonnage hybride: Si l'objet d'étalonnage n'est pas tridimensionnel mais plan ou cylindrique, il faut recourir à l'étalonnage hybride pour fixer de façon explicite une partie des paramètres de la caméra (ceux de la matrice K) et identifier sous forme implicite ceux de la matrice T par auto-étalonnage.

D. Etude pratique d'un logiciel permettant de faire cet étalonnage

La calibration de la caméra peut se faire de plusieurs manières.

Une des plus connues utilisée sur openCV consiste à prendre un damier de taille connu, le capturer avec différents points de vue grâce à la caméra et trouver les coordonnées de pixel des points 3D des images grâce à la fonction findChessboardCorners().

Enfin, la méthode calibrateCamera() permet de trouver les paramètres de notre caméra.

Elle nous donnera la matrice de la camera, le coefficient de distorsion et les vecteurs de rotation et de translation.

Voici un résultat typique d'une image de calibration via openCV :

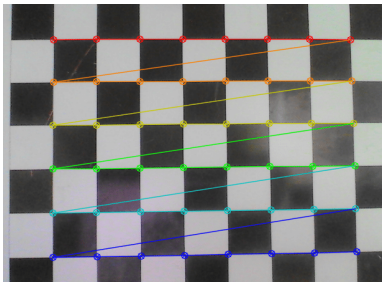

Fig. 6. Détection des coins par openCV

IV. Détection des obstacles et du robot

Il est essentiel de détecter les obstacles et les robots présents sur le terrain afin de pouvoir effectuer des traitements sur ces derniers. Plusieurs solutions sont disponibles, et nous avons sélectionné celles qui nous semblaient les plus pertinentes.

A. Détection du robot

Pour identifier le robot, nous allons utiliser des codes ArUco que nous placerons au centre de celui-ci, de manière à ce qu'ils soient facilement visibles depuis la caméra qui sera placée en hauteur.

Fig. 7. Exemple de code ArUco

Ils ressemblent à des QR codes, mais ils contiennent beaucoup moins d'informations et ont donc moins de détails. Cela permet d'avoir une résolution plus faible sur la caméra, car l'information est plus facilement lisible de loin et de mauvaise qualité. Cela nous permet de positionner la caméra en retrait et d'obtenir ainsi un champ de vision beaucoup plus large qu'avec une autre solution.

En effet, la résolution nécessaire pour récupérer l'information est beaucoup moins importante. Contrairement au QR code qui possède des informations redondantes pour ne pas avoir d'orientation, le code ArUco a une orientation définie, ce qui nous sera utile par la suite.

1) Détection ArUco: Pour détecter les codes ArUco, nous allons utiliser la bibliothèque OpenCV et son module ArUco.

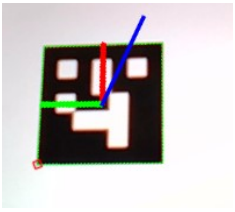

Fig. 8. Exemple détection de code ArUco

Une fois le code détecté, il nous donne une orientation, ce qui nous permet de déduire l'orientation du robot auquel le code est rattaché.

Premièrement il faut récupérer les axes qui sont pertinentes

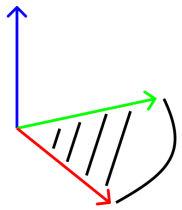

Fig. 9. Axes 3

Nous avons utilisé la bibliothèque OpenCV pour obtenir des vecteurs de rotation. Cependant, pour faires des calculs nous devons convertir ces vecteurs en matrices de rotation. Pour ce faire, nous avons utilisé la méthode de Rodrigues, qui permet de dériver une matrice de rotation à partir d'un vecteur de rotation donné.

La méthode de Rodrigues est basée sur la formule suivante :

$$R = I + sin(\theta)S + (1 - cos(\theta))S^2$$

où R est la matrice de rotation, I est la matrice identité, θ est la norme du vecteur de rotation et S est la matrice antisymétrique associée au vecteur de rotation.

Une fois la matrice de rotation obtenue, nous avons pu récupérer les angles d'Euler correspondants en utilisant la fonction cv2.decomposeProjectionMatrix de OpenCV. Les angles d'Euler représentent les rotations autour des trois axes de l'espace et peuvent être utilisés pour analyser les angles.

Dans notre cas, nous nous sommes intéressés à l'angle de rotation autour de l'axe vertical, qui correspond à la rotation de l'objet autour de son axe vertical.

Pour adapter les valeurs de l'angle à notre problème spécifique, il est nécessaire de les recentrer.

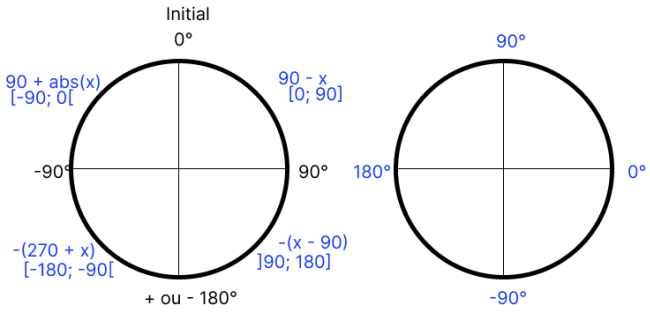

Fig. 10. Transformation de la valeur des angles

V. Planification de trajectoire

A. Les algorithmes de planification de trajectoire

Afin que notre robot puisse aller d'un point A à un point B, nous avons décidé d'utiliser un algorithme de recherche de chemin basé dans un repère. Voici un état de l'art de tous les algorithmes existants. Les illustrations de cette partie proviennent du site PythonRobotics.

1) Contexte: Nous nous plaçons dans l'ensemble des états suivant : des noeuds reliés entre eux par des arcs unidirectionnels ou bidirectionnels. Chaque noeud est caractérisé par ses coordonnées x et y. Chaque noeud possède un ensemble de prédécesseurs (il existe un arc allant du noeud prédécesseur au noeud étudié) et un ensemble de successeurs (il existe un arc allant du noeud étudié au noeud successeur). Les algorithmes ci-dessous permettent de trouver un chemin permettant d'aller d'un noeud de départ à un noeud d'arrivée.

2) L'algorithme de parcours en largeur: Cet algorithme permet le parcours récursif d'un graphe ou d'un arbre à partir d'un nœud source. Le principe du parcours en largeur est le suivant : on explore en premier le nœud source, puis ses successeurs, puis les successeurs de ses successeurs, etc. Cet algorithme visite tous les nœuds du graphe atteignable à partir du nœud source. Il peut être utilisé afin de calculer les plus courts chemins entre le sommet source et tous les sommets du graphe. Sa complexité temporelle dans le pire cas est O(s+a) avec s le nombre de sommets et a le nombre d'arcs.

La figure 11 illustre une recherche de chemin avec l'algorithme de parcours en largeur. Les croix bleues représentent les points qui ont été parcourus par l'algorithme, le point vert représente le point de départ, et la ligne rouge le chemin obtenu.

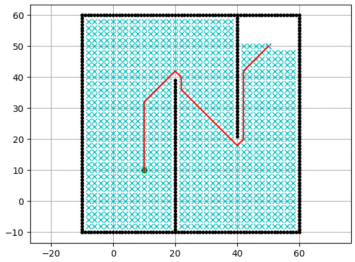

Fig. 11. Recherche de chemin avec l'algorithme de parcours en largeur

3) L'algorithme de parcours en profondeur: Le parcours en profondeur est un algorithme récursif de parcours de graphe ou d'arbre. A partir d'un sommet source, il explore un chemin dans le graphe jusqu'à arriver à une impasse ou à un nœud déjà visité. Il revient alors au dernier sommet pouvant mener à un autre chemin et explore celui-ci. L'algorithme s'arrête lorsque tous les sommets atteignables à partir du sommet source ont été visités. Il est utilisé afin de trouver

l'ensemble des sommets accessibles depuis un sommet source. Sa complexité dans le pire des cas est O(s+a) avec s le nombre de sommets et a le nombre d'arcs.

La figure 12 illustre une recherche de chemin avec l'algorithme de parcours en profondeur. Les croix bleues représentent les points qui ont été parcourus par l'algorithme, le point vert représente le point de départ et la ligne rouge le chemin obtenu.

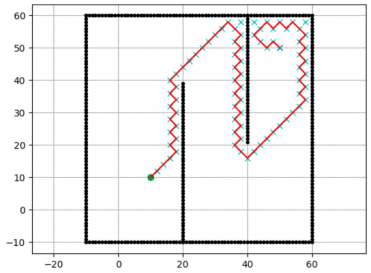

Fig. 12. Recherche de chemin avec l'algorithme de parcours en profondeur

4) L'algorithme de Dijkstra: Cet algorithme est un algorithme de recherche du plus court chemin dans un graphe pondéré positivement. Il est fondé sur un parcours en largeur. A partir d'un point de départ, il recherche le plus court chemin menant à la destination de façon itérative. L'algorithme se base sur le principe que tous les sous-chemins C-D du chemin le plus court entre les points A et B sont aussi le plus court chemin entre les points C et D. La complexité temporelle de l'algorithme de Dijkstra varie en fonction de la manière dont il est implémenté : avec un tas binaire, elle est de O((a+s) x log(s)) ; avec un tas de Fibonacci, elle est de O(a+s x log(s)).

La figure 13 illustre une recherche de chemin avec l'algorithme de Dijkstra. Les croix bleues représentent les points qui ont été parcourus par l'algorithme, le point vert représente le point de départ et la ligne rouge le chemin obtenu.

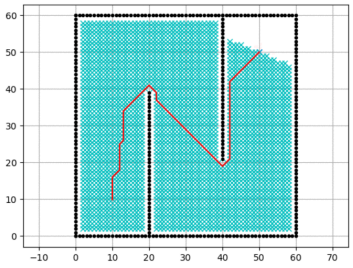

Fig. 13. Recherche de chemin avec l'algorithme de Dijkstra

5) L'algorithme A:* L'algorithme A* est un algorithme de recherche du plus court chemin d'un point à un autre dans un graphe. C'est est une version améliorée de l'algorithme de Dijkstra qui utilise des heuristiques pour orienter sa recherche du plus court chemin. Les heuristiques estiment la distance qui sépare le point considéré du point d'arrivée. Pour que l'algorithme soit admissible, c'est-à-dire pour qu'il garantisse de trouver le plus court chemin, l'heuristique choisie ne doit pas surestimer la distance au point d'arrivée. La complexité de l'algorithme dépend de la qualité de l'heuristique utilisée. L'un des avantages de A* est sa simplicité de calculs. Toutefois, celui-ci ne peut pas adapter le chemin trouvé si l'environnement (le graphe) change.

La figure 14 illustre une recherche de chemin avec l'algorithme A*. Les croix bleues représentent les points qui ont été parcourus par l'algorithme, le point vert représente le point de départ et la ligne rouge le chemin obtenu.

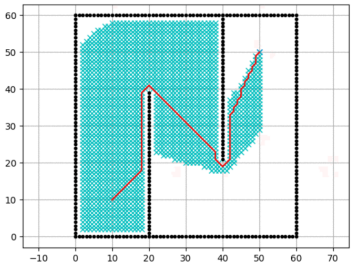

Fig. 14. Recherche de chemin avec l'algorithme A*

6) L'algorithme D ou l'algorithme A* dynamique:* L'algorithme D* ou l'algorithme A* dynamique. Cet algorithme se comporte comme A* avec la différence que l'algorithme permet une replanification rapide de la trajectoire si un obstacle inattendu est détecté sur la route. L'algorithme commence à étudier les nœuds à partir du nœud d'arrivée. Chaque nœud possède un "backpointer" qui représente le nœud étudié précédemment (donc le prochain nœud de la trajectoire finale), et le coût pour aller d'un nœud au nœud d'arrivée est connu. L'algorithme se termine lorsque le nœud de départ est rencontré. Le chemin à suivre est alors indiqué par les "backpointers".

La figure 15 illustre une recherche de chemin avec l'algorithme A*. Les croix bleues représentent les points qui ont été parcourus par l'algorithme, le point vert représente le point de départ et la ligne rouge le chemin obtenu.

7) LPA ou Lifelong Planning A*:* Cet algorithme est une version incrémental de A* qui réutilise des informations calculées lors de précédentes recherches. Sa première recherche est l'algorithme A*. L'algorithme trouve de manière répétée les chemins les plus courts d'un nœud de départ à un nœud d'arrivée dans un graphe, à mesure que des arcs ou des nœuds sont ajoutés ou supprimés ou que les coûts des arêtes sont modifiés. LPA* permet de réduire fortement

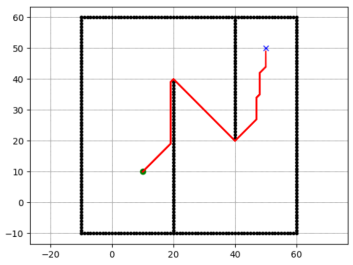

Fig. 15. Recherche de chemin avec l'algorithme D*

la durée de recherche lorsque des changements mineurs surviennent dans le graphe. En effet, l'algorithme ne réévalue que les noeuds affectés par le changement, contrairement à A*.

8) D Lite:* Cet algorithme est un algorithme de recherche heuristique incrémentale qui a le même comportement que l'algorithme D* mais qui se base sur le LPA*. Contrairement au LPA* qui commence sa recherche à partir du noeud de départ, D* Lite commence à chercher à partir du nœud d'arrivée. Cela permet la réutilisation des valeurs calculées lors d'une replanification, même quand le point de départ change. Cet algorithme, plus simple à implémenter que D* et a une performance égale voire supérieure à celui-ci.

La figure 16 illustre une recherche de chemin avec l'algorithme D*. La figure 17 représente la replanification de chemin que permet l'algorithme lorsqu'un obstacle apparaît sur le chemin initial.

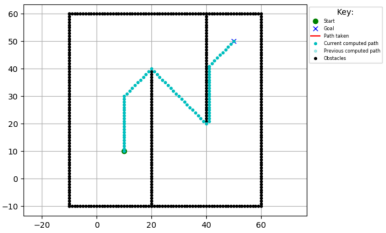

Fig. 16. Recherche de chemin avec l'algorithme D* Lite

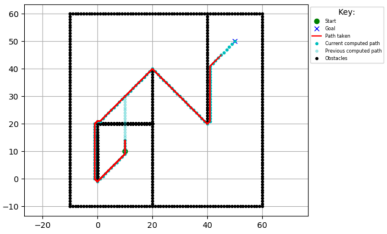

Fig. 17. Replanification de chemin avec l'algorithme D* Lite

B. L'algorithme choisi

Dans l'étude de notre robot, A* peut représenter un bon choix en raison de sa simplicité de calcul. Toutefois, l'algorithme présente un inconvénient : il est incapable de s'ajuster lorsque des obstacles inattendus apparaissent sur le chemin. Il n'est donc pas adapté à notre situation. Parmi les algorithmes de recherche incrémentale, D*, LPA* et D* Lite, nous avons décidé d'utiliser l'algorithme D* Lite, de par sa simplicité d'implémentation et son efficacité.

La figure 18 représente le pseudo-code de l'algorithme D* Lite que nous avons utilisé. Cet algorithme provient du papier "Improved Fast Replanning for Robot Navigation in Unknown Terrain".

```
procedure CalculateKey(s)
{01"} return [min(g(s), rhs(s)) + h(s_start, s)+k_m; min(g(s), rhs(s))];
procedure Initialize()
{02"} U := ∅;
{03"} k_m = 0;
{04"} for all s ∈ S rhs(s) = g(s) = ∞;
{05"} rhs(s_goal) = 0;
{06"} U.Insert(s_goal, CalculateKey(s_goal));
procedure UpdateVertex(u)
{07"} if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
{08"} if (u ∈ U) U.Remove(u);
{09"} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{10"} while (U.TopKey() < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
{11"}     k_old = U.TopKey();
{12"}     u = U.Pop();
{13"}     if(k_old < CalculateKey(u))
{14"}         U.Insert(u, CalculateKey(u));
{15"}     else if (g(u) > rhs(u))
{16"}         g(u) = rhs(u);
{17"}         for all s ∈ Pred(u) UpdateVertex(s);
{18"}     else
{19"}         g(u) = ∞;
{20"}         for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{21"} s_last = s_start;
{22"} Initialize();
{23"} ComputeShortestPath();
{24"} while (s_start ≠ s_goal)
{25"}     /* if (g(s_start) = ∞) then there is no known path */
{26"}     s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
{27"}     Move to s_start;
{28"}     Scan graph for changed edge costs;
{29"}     if any edge costs changed
{30"}         k_m = k_m + h(s_last, s_start);
{31"}         s_last = s_start;
{32"}         for all directed edges (u, v) with changed edge costs
{33"}             Update the edge cost c(u, v);
{34"}             UpdateVertex(u);
{35"}         ComputeShortestPath();
```

Fig. 18. Pseudo-code de D* Lite

C. L'espace des états considéré par l'algorithme choisi

Nous considérons une grille de deux dimensions comme notre espace d'états. Elle sera détaillée dans la section suivante "Représentation de l'environnement". Chaque case de cette grille représente un état de l'espace de recherche, et chaque état, appelé noeud, est caractérisé par ses coordonnées x et y. Un noeud est donc de la forme suivante : (x, y). Chaque noeud possède 2 à 8 voisins : les noeuds se situant dans les coins de la grille possèdent 3 voisins (figure 21), ceux se situant sur les bords possèdent 5 voisins (figure 20), et les autres en possèdent 8 (figure 19).

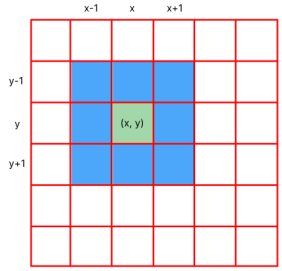

Fig. 19. Voisins (bleu) d'un noeud (vert) se situant à l'intérieur de la grille

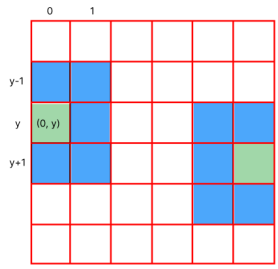

Fig. 20. Voisins (bleu) d'un noeud (vert) se situant sur le bord de la grille

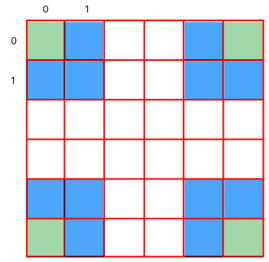

Fig. 21. Voisins (bleu) d'un noeud (vert) se situant dans un coin de la grille

D. Représentation de l'environnement

La première étape de la planification consiste à extraire l'information présente dans l'image pour la rendre exploitable pour notre algorithme de planification.

La solution la plus facile serait de prendre l'image et de considérer chaque pixel comme un nœud. Cependant, cette solution entraîne des problèmes de performances. En prenant une image de taille 1000x1000, on obtient plus de 1 000 000 de nœuds. Dans ces conditions, l'algorithme de planification sera lent.

De plus, peut-on réellement considérer un pixel comme un nœud à part entière, compte tenu de la taille de celui-ci ? Cela n'est guère pertinent car un pixel ne représente presque rien.

De ce fait, nous pratiquons une discrétisation de l'image. Chaque nœud sera un groupe de pixels, ce qui permet d'avoir une performance meilleure en réduisant la taille de l'espace de recherche, ainsi que de choisir une taille modulaire en lien avec le rapport pixel/mm de notre arène.

Maintenant que l'image est discrétisée, nous avons une base sur laquelle caler notre discrétisation qui représentera la modélisation de notre problème. De plus, comme la grille sera superposée à l'image sur laquelle nous allons détecter les différents éléments pertinents (robot, obstacles), il est facilement possible de remplir les nœuds de l'état qui les concernent et d'obtenir l'environnement sur lequel appliquer notre algorithme de planification.

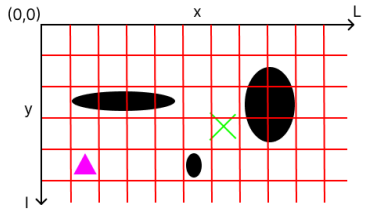

Fig. 22. Discrétisation

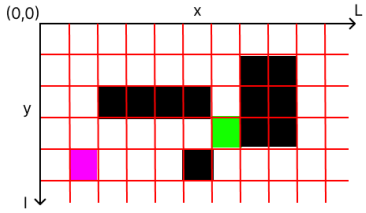

Fig. 23. Résultat final

E. Amélioration

La gestion des obstacles telle que présentée ci-dessus est cependant trop rigide, car elle ne prend pas en compte les faux positifs, où un pixel pourrait être détecté comme un obstacle alors qu'il n'y en a pas réellement, ce qui entraînerait la considération de toute la case comme un obstacle.

De plus, pour garantir que le robot puisse réellement traverser, nous allons artificiellement augmenter la taille des obstacles de la taille du robot. Ainsi, dans notre image, nous appliquerons une dilatation de nos obstacles de la taille du robot.

1) Faux positif: Pour garantir la présence réelle d'un obstacle dans une case, nous vérifions que plus de 5% de la case est effectivement détectée comme un obstacle avant de la considérer comme tel.

2) Dilatation des obstacles: La dilatation des obstacles est une étape cruciale pour assurer le passage du robot dans des espaces restreints. Pour ce faire, nous dilatons chaque obstacle de la moitié de la taille du robot. Voici deux exemples illustrant l'importance de cette technique :

Dans la figure 24, le robot dispose de l'espace nécessaire pour effectuer une transition en diagonale. La dilatation préalable garantit que le robot peut passer sans encombre.

En revanche, dans la figure 25, le robot se trouve confronté à un espace trop restreint. La dilatation des obstacles détecte cette contrainte et empêche ainsi le déplacement du robot.

Il est clair que la dilatation empêche la transition en diagonale lorsque le robot ne dispose pas de l'espace

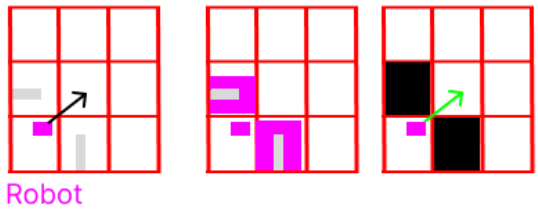

Fig. 24. Robot avec suffisamment d'espace pour traverser.

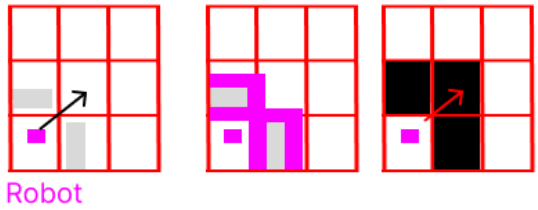

Fig. 25. Robot sans suffisamment d'espace pour traverser.

nécessaire pour passer.

Le dernier cas à gérer est la gestion des bords. En effet, on suppose que la limite de l'arène est inaccessible. Il faut donc bloquer tous les bords de l'arène et les considérer comme des obstacles pour ensuite les dilater.

F. Taille du robot

Pour réaliser cela, il est essentiel de déterminer la taille en pixels du robot. Afin de rendre notre solution flexible, nous proposons une méthode permettant d'obtenir rapidement cette valeur, évitant ainsi la saisie manuelle, qui peut être fastidieuse.

Pour ce faire, nous utilisons un objet témoin, tel qu'un cercle dont nous connaissons la taille. Celui-ci est placé sous la caméra, puis nous utilisons la méthode de HoughCircles qui détecte le cercle et nous fournit le rayon du cercle en pixels.

Étant donné que nous connaissons le diamètre en pixels et sa taille réelle, nous pouvons calculer le ratio pixel/taille de notre environnement. Ensuite, connaissant la taille réelle du robot, il suffit d'appliquer ce ratio à sa taille pour obtenir sa taille en pixels.

G. Résultat de la planification

Suite à la planification avec l'algorithme D* Lite, nous obtenons un chemin composé de tous les noeuds (x, y) que le robot doit suivre pour atteindre le noeud d'arrivée.

Pour pouvoir faciliter l'étape suivante du projet, le suivi de trajectoire, nous avons décider de transformer le chemin obtenu en un chemin de noeuds (x, y, theta) où theta représente l'angle à laquelle doit se situer le robot afin d'aller à la prochaine case prévu sur le chemin. Cet angle représente la "direction" que doit prendre le robot. Situé dans une case (x, y) de la grille, il y a 8 angles possibles, chacun pour se diriger vers un voisin possible de l'angle (x, y). La figure 26 montre les différents angles possibles d'un noeud (x, y) en fonction du prochain noeud sur la trajectoire. Le dernier noeud du chemin, c'est-à-dire le noeud d'arrivée, prend l'angle du noeud du chemin le précédant.

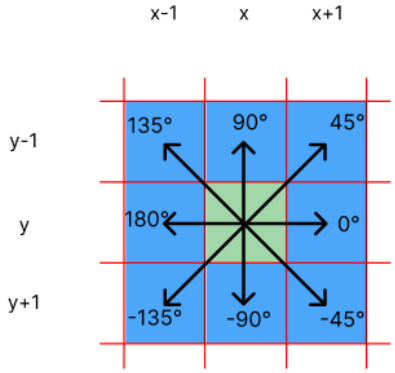

Fig. 26. Angles pouvant prendre le noeud (x, y) pour aller à un voisin

VI. Suivi de trajectoire

A. Calcul de l'orientation

Afin de suivre la trajectoire du robot, il est essentiel de pouvoir calculer la rotation nécessaire pour qu'il se dirige vers ses objectifs. Pour ce faire, nous pouvons modéliser le problème de la manière suivante.

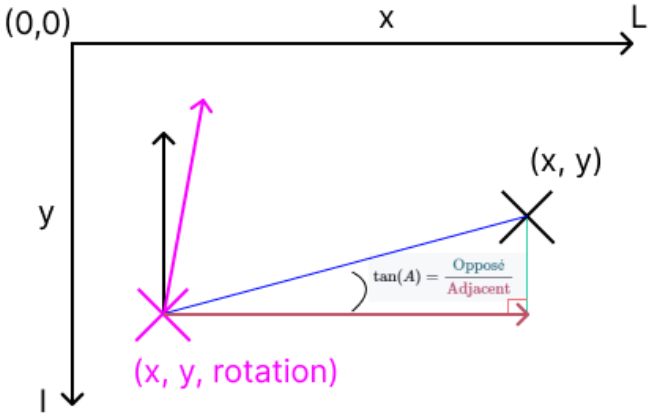

Fig. 27. Modélisation du problème du calcul de l'orientation

Il est nécessaire de prendre en compte les différents cas possibles lorsque le point (x,y) se situe en dessous du robot ou à sa gauche. En effet, cela permet de définir quatre quadrants distincts.

Le premier quadrant est déjà pris en compte dans la modélisation du problème, si l'on se place dans notre référentiel d'angles (voire figure 10) Nous pouvons récupérer

la valeur correspondante de manière efficace en procédant comme suit:

(angle = valeur calculée par le premier quadrant)

1) Quadrant haut gauche:

$$res = (angle - 180) * -1$$

2) Quadrant bas gauche:

$$res = angle - 180$$

3) Quadrant bas droite:

$$res = angle * -1$$

B. Fonction de rotation

Une fois les différentes orientations calculées, il est nécessaire de pouvoir les suivre. Pour cela, différentes fonctions permettent de s'aligner avec les huit positions angulaires possibles. Leurs vitesses diffèrent pour que la rotation soit un compromis entre rapidité et précision.

C. Fonction de suivi en ligne droite

Ensuite, quand le robot est positionné dans la bonne orientation, il convient de le faire avancer tout droit pour passer à la case suivante. Le robot possédant un moteur droit et un moteur gauche, il a fallu les tester pour convenir de leur vitesse respective pour avancer en ligne droite. Aucun écart entre les deux n'ayant été observé sur les petites distances nécessaires à l'algorithme, la même vitesse peut être adoptée pour les deux moteurs afin d'aller tout droit.

D. Algorithme de suivi de trajectoire

Enfin, l'algorithme contient différentes étapes. Tout d'abord la récupération des coordonnées de chaque étape de la trajectoire envoyée par l'algorithme de planification. Il convient donc de chercher les coordonnées successives en x, en y et les orientations. Une fois ces dernières obtenues, chaque étape est composée de deux mouvements : premièrement, la rotation pour s'aligner avec l'angle souhaité; deuxièmement, l'avancée en ligne droite pour changer de case. Ces actions répétées doivent permettre d'atteindre la fin de la trajectoire. En pratique, il faut se connecter au robot en créant un singleton, et utiliser son attribut set_var qui fixe une vitesse dans les motor.left.target et motor.right.target ainsi qu'un time.sleep d'une certaine durée pour aller au bout du mouvement. La connexion unique via le singleton permet d'enchaîner les commandes et de ne déconnecter le robot qu'une fois l'objectif atteint.

VII. L'ARÈNE DU ROBOT

Dans cette partie, nous allons suivre les étapes que nous suivons pour initialiser l'arène du robot pour permettre une première planification de trajectoire.

La première étape après avoir installé le matériel est de détecter l'arène sur laquelle nous allons travailler. Celle-ci est délimité par deux codes Aruco. La figure 28 montre une photo de notre arène délimité par les codes Aruco.

Fig. 28. Arène délimitée par deux codes Aruco

Ensuite, nous devons calibrer l'image afin de pouvoir récupérer la taille du robot. Pour cela, nous utilisons un jeton rouge. La figure 29 montre la détection du jeton.

Fig. 29. Calibration de l'image avec la détection du jeton

La calibration permet alors de récupérer la taille du robot, représenté par un code Aruco), afin de permettre la détection et la dilatation des obstacles rouges, comme expliqué dans les parties précédentes. La planification de l'arène se fait à partir de cela.

La figure 30 montre la discrétisation de l'arène ainsi que la détection des obstacles rouges. Les cases rouges sont les cases "obstacles" auquels le robot ne peut pas accéder. Dans notre image, une discrétisation 100x100 a été réalisée (100 cases sur la largeur et 100 sur la longueur).

Les figure 31 et figure 32 montrent la trajectoire issue de l'algorithme de planification.

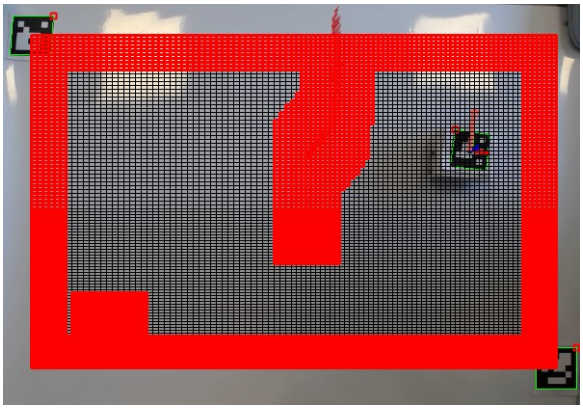

Fig. 30. Discrétisation de l'arène et détection des obstacles rouges

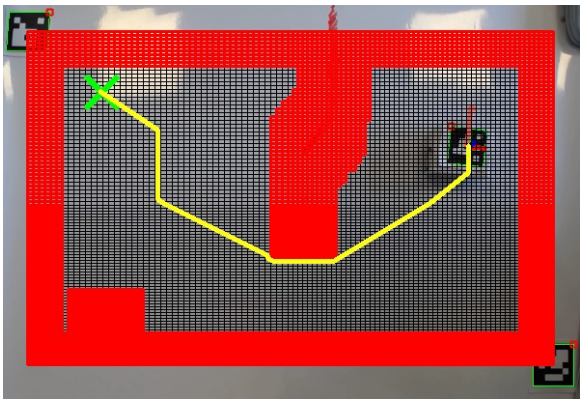

Fig. 31. Trajectoire planifiée avec discrétisation

Fig. 32. Trajectoire planifiée

VIII. Conclusion

A. Résultats obtenus

Notre objectif était de communiquer avec un robot dans une arène afin de lui faire suivre une trajectoire planifiée. A la fin du projet, nous sommes capables de détecter une arène grâce à deux codes ArUco, calibrer la caméra et trouver sa résolution en direct. Le robot Thymio est également étecté grâce à un code ArUco et nous pouvons retrouver les obstacles sur son chemin dilatés en rouge. Une fois le point d'arrivée défini, nous pouvons planifier une trajectoire et assurer son suivi. Des pistes d'amélioration seraient de faire un trajet en continu avec des courbes au lieu de décomposer le mouvement en deux étapes (rotation puis ligne droite).

B. Bilan global

Ce projet nous aura apporté des connaissances en robotique, en traitement d'images et en planification et suivi de trajectoire. D'une part, nous avons pu nous documenter pour choisir les meilleures solutions à mettre en pratique sur un cas concret. D'autre part, nous avons pu coder les différentes étapes de ce projet et les mettre en place au fur et à mesure de nos avancées. De plus, ce projet nous aura permis de découvrir le monde de la recherche qui se base sur des problématiques concrètes et cherche ses solutions en terme d'efficacité et de coût.

REMERCIEMENT

Un grand remerciement à M. Boumaza qui nous a encadrés, accueillis dans ses locaux et conseillés de manière pédagogique. Grâce à lui, cette expérience a été très enrichissante. Nous tenons également à remercier Télécom Nancy pour être à l'origine de cette collaboration, ainsi que les lecteurs de cet article pour leur attention et le temps accordé.

REFERENCES

[1] Mordechai Ben-Ari, Francesco Mondada, *Elements of Robotics* Livre sur SpringerLink, 2018.
[2] G.W. Lucas *trajectory Model for the Differential Steering System* Rossum project, 2000.
[3] Udek and Jenkin *Computational Principles of Mobile Robotics* Université de Colombia
[4] O. Faugeras *Three-dimensional computer vision: a geometric viewpoint* Cambridge, MA (USA) : MIT Press, 1993.
[5] Tim Wescott *PID Without a PhD* Wescott Design Services, 2018
[6] Iconnue *Kinematics and Odometry for a Differential Drive Vehicle* California Institute of Technology
[7] Lulu *Modèle géométrique d'un robot mobile à roues différentielles* Le blog de Lulu, 2019