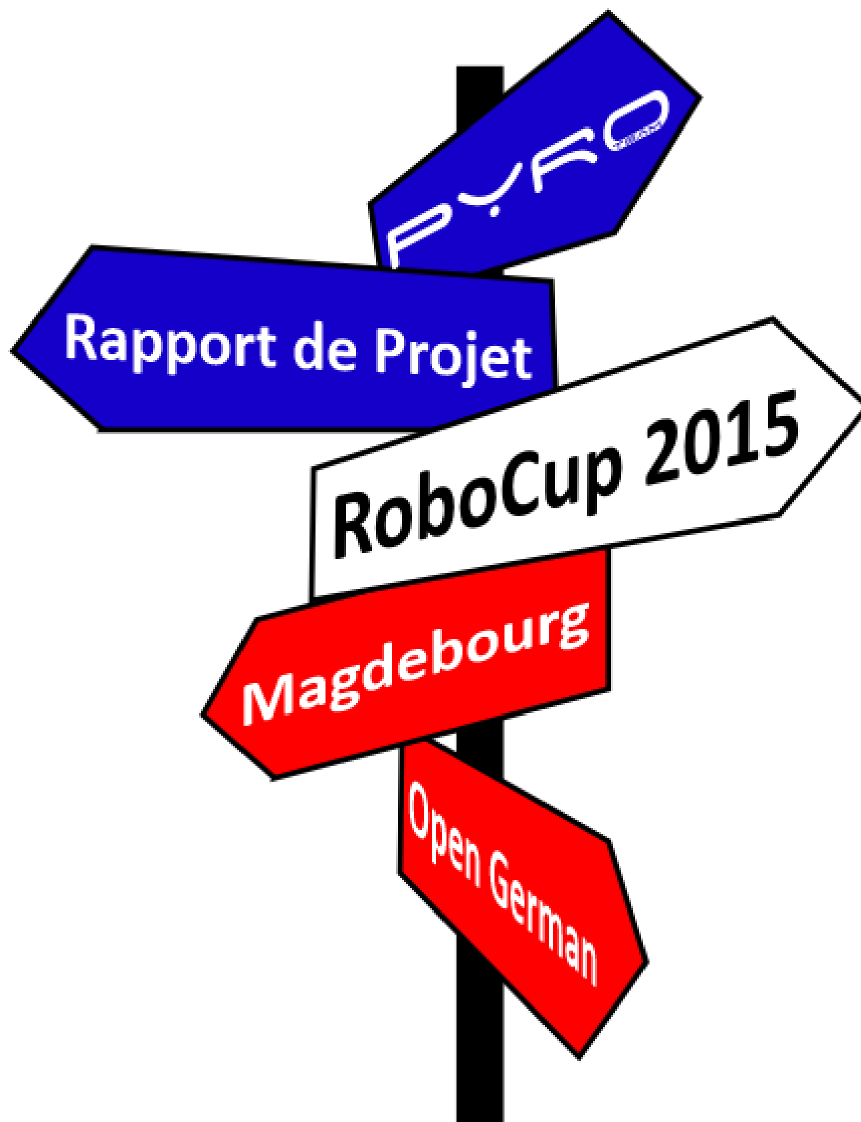
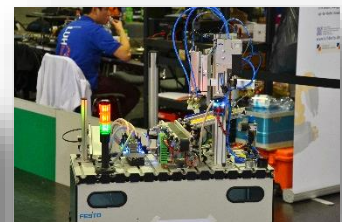




Rédigé par Thomas DANEL et Romain KRIKORIAN

Projet encadré par Rochdi MERZOUKI et Vincent COELEN

Remerciements

Nous tenons à remercier le département IMA pour l'investissement financier fourni afin de payer les différents frais de déplacement ainsi que ceux concernant le logement à Magdebourg en Allemagne.

Nous remercions le centre de recherche CRIStAL de nous prêter l'un de ses locaux. Ainsi l'ensemble de l'équipe a pu réaliser de nombreux tests afin de valider les différentes conceptions matérielles et logicielles.

Nous tenons à remercier M. Merzouki pour nous avoir fait confiance en nous proposant ce projet afin de préparer l'Open German 2015, compétition qui nous a permis de mettre en pratique nos compétences et d'en acquérir de nouvelles. Aussi, il nous a épaulés dans la recherche de sponsors en particulier auprès de l'université Lille 1.

Aussi, nous remercions Mr Coelen, le « team leader » de l'équipe, pour nous avoir suivis et épaulés tout au long de ce projet, tant du point de vue technique qu'organisationnel. Nous avons beaucoup appris de son expérience dans le domaine de la robotique mobile et dans la gestion de projet de manière plus générale.

Nous tenons à remercier les organisateurs de la compétition, ainsi que FESTO, l'entreprise partenaire de la Logistic League, pour la qualité de leurs services et la mise en place de l'événement.

Pour finir, nous tenons aussi à remercier toute l'équipe Pyro Team : Vincent Coelen, Valentin Vergez, Elise Tissot, Sandra Hage Chehade, Cyril Smagghe et Vianney Payelle. Sans tous les membres de l'équipe nous n'aurions pas pu aller si loin, 700 kms ce n'est pas rien.

Sommaire

Remerciements 1
Introduction 3
1) Contexte 3
2) Cahier des charges 3
I- le Projet 4
1) Objectif 4
2) Choix techniques 4
a. Matériel 4
b. Logiciel 4
3) Etapes du projet 5
4) Organisation du travail 5
II- les Résultats 7
1) Landmarks extraction 7
a. Filtrage 7
b. Reconnaissance de droites 7
c. Extraction des machines 9
2) Correction de l'odométrie 10
3) Extended Kalman Filter 10
a. Data Association 10
b. Algorithme de Kalman 11
4) Grid Map 15
a. Génération de map vide 15
b. Prise en compte des obstacles fixes (machines) 17
5) Pathfinder 18
a. Choix de l'algorithme 18
b. L'algorithme A 19*
6) Path tracker 23
Conclusion 24
Annexe 25

Introduction

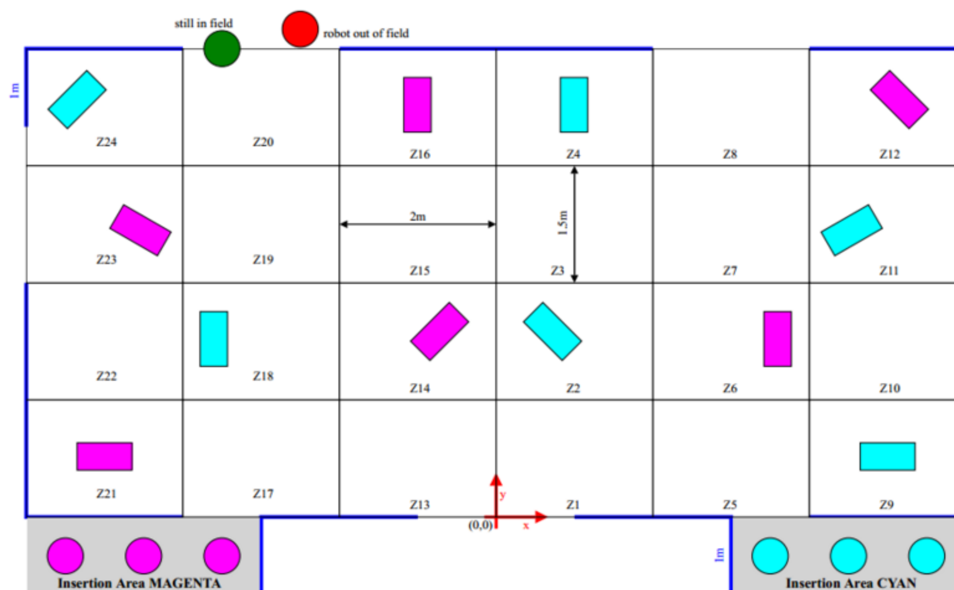
1) Contexte

Dans le cadre de la participation de Polytech Lille dans une compétition internationale de robotique, la RoboCup, nous avons réunis plusieurs étudiants de la spécialité IMA afin de créer une équipe : la Pyro TEAM (PolYtech RObocup TEAM). Il y avait déjà une équipe l'année dernière, et c'est dans la continuité du travail réalisé par les anciens membres que l'équipe actuelle a participé, et participera à la RoboCup dans la Logistics League.

2) Cahier des charges

Le cahier des charges fixé par le règlement de la compétition demande aux équipes de fournir des solutions pour réaliser des actions demandées par un arbitre : la Referee Box.

Nous traiterons dans ce projet de l'aspect navigation des robots, composé d'une partie **localisation** et d'une partie **déplacement**. L'environnement est connu, mais l'emplacement des machines est aléatoire, fixé par la Referee Box en début de partie.



La navigation sera donc utilisée dans la **phase d'exploration** afin de réaliser une carte de la zone de jeu permettant de définir des zones de passage entre les machines, ainsi que l'emplacement exact de celles-ci.

Dans la **phase de production**, la navigation permettra de se déplacer à partir de la carte créée au préalable avec les obstacles fixes et des robots (obstacles mobiles) se déplaçant en même temps.

I- le Projet

1) Objectif

L'objectif général est de fournir aux Robotinos un système capable de se localiser et de parcourir des trajectoires calculées à partir de coordonnées envoyées par le manager. Plus précisément, les différentes parties à traiter seront :

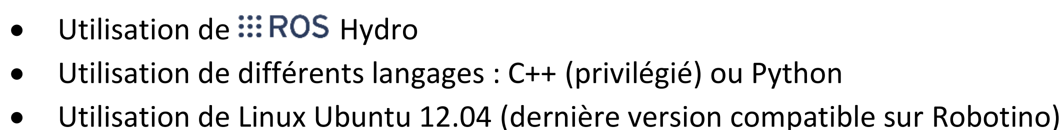
- Localiser correctement le robot (à 5 cm près)
- Localiser les éléments fixes (Murs, Machines)
- Générer une trajectoire selon les demandes du manager et la détection d'obstacles dynamiques (robots)
- Assurer le suivi de la trajectoire

2) Choix techniques

a. Matériel

- Utilisation de 3 Robotinos équipés chacun de :
    - 1 détecteur laser pouvant réaliser des mesures à 240°
    - 1 gyroscope
    - 9 capteurs SHARP (télémètres infrarouges)
    - 3 codeurs incrémentaux présents en sortie de chaque moteur du Robotino

b. Logiciel



- Utilisation de ROS Hydro
- Utilisation de différents langages : C++ (privilégié) ou Python
- Utilisation de Linux Ubuntu 12.04 (dernière version compatible sur Robotino)

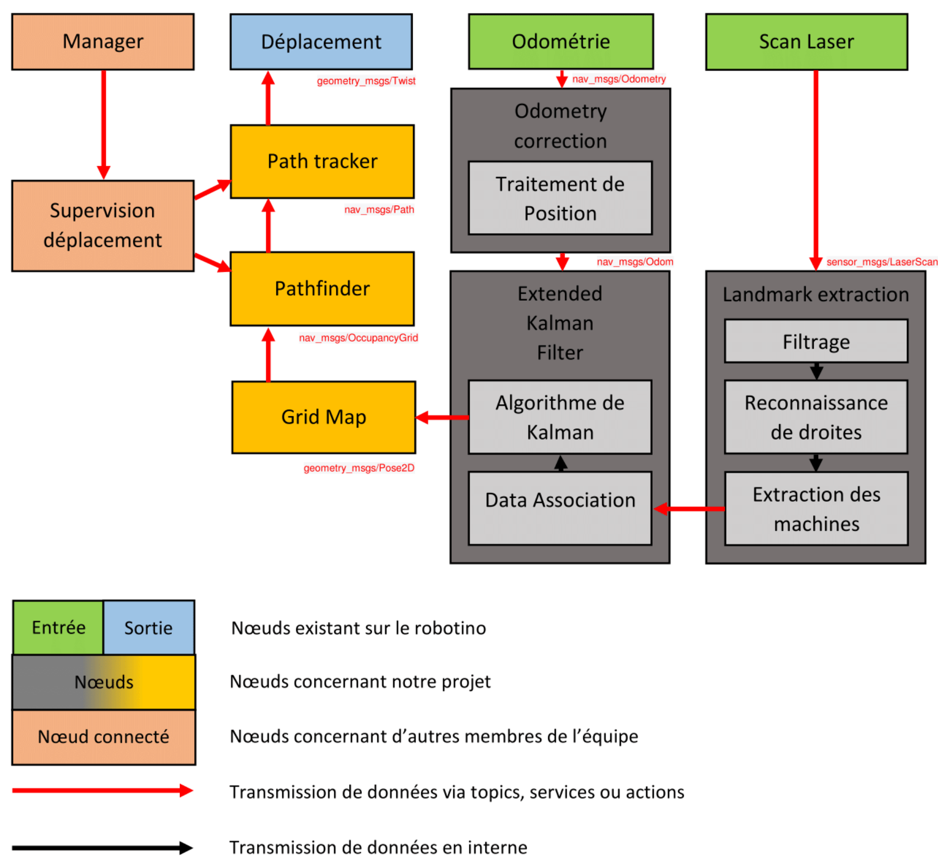
3) Etapes du projet

- Conception du schéma global des différentes parties :
  - Localisation avec le SLAM
  - Création de la carte avec le SLAM
  - Génération de trajectoires avec A*
  - Exécution de trajectoire
- Codage des différents nœuds ROS
- Tests de précision et de robustesse sur robot
- Validation du modèle

4) Organisation du travail

Le schéma suivant représente le travail à réaliser, réparti entre nous. A savoir, les nœuds en gris correspondent au travail de Thomas, les nœuds en jaune à celui de Romain.

Schéma final du projet



Décrivons précisément le rôle de chaque nœud :

Scan Laser et *Odométrie* sont les deux topics sur lesquels on a besoin de se connecter pour récupérer les données brutes du laser d'un côté, et récupérer la position du robot par rapport au point de départ. Il renvoie une position corrigée grâce au gyroscope (Pose) et une vitesse non corrigée par le gyroscope (Twist).

L'*Odométrie* n'étant pas corrigée, le nœud **Odometry correction** corrige l'erreur sur le Twist avec les données du gyroscope (par dérivation).

Dans la partie **Landmarks Extraction**, on a plusieurs fonctions :

- *Filtrage* permet d'enlever les éventuels bruits qui affectent les données du laser
- *Reconnaissance des droites* traite les données laser filtrées pour détecter les droites
- *Extraction des machines* extrapole les objets connus (machines) à partir des droites.

Pour la partie **EKF**, la fonction *Data Association* lie la position des machines détectées au laser à leur position dans le repère global. Le passage dans ce repère se fait par rapport à la position calculée du robot. Ensuite, l'*algorithme de Kalman* va nous donner les positions de chaque machine observée, ainsi que la position du robot par rapport à celles-ci. C'est le nœud central du SLAM.

Le nœud **Manager** envoie la position à atteindre au nœud **Supervision déplacement**, qui va faire son possible pour que le robot aille à la position demandée, en fonction des positions machines et des obstacles mobiles.

Grid Map renvoit une map utilisée ensuite pour les déplacements et les calculs de trajectoires

Pathfinder recherche le chemin demandé par le Manager sur une grille, il est basé sur l'algorithme A star.

Path tracker exécute le chemin trouvé en fournissant les vitesses au nœud **Déplacement** (nœud robotino_local_move_server), qui donne les commandes aux moteurs.

II- les Résultats

1) Landmarks extraction

a. Filtrage

Le filtrage effectué est juste un filtrage sur les distances données par le laser, toutes les distances < 0 et > 5 m sont supprimées. Sachant que toutes les données laser sont bruitées au maximum de 3% de la mesure.

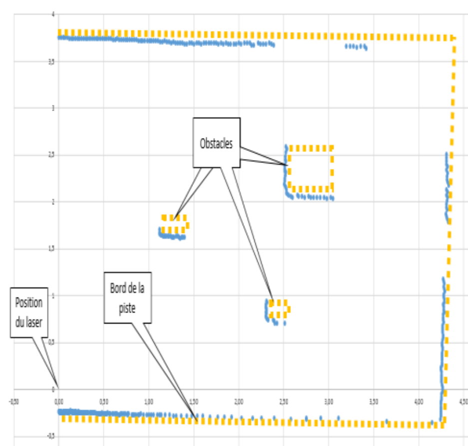
b. Reconnaissance de droites

Pour la reconnaissance de droites, l'algorithme choisi est RANSAC, abréviation pour RANdom SAmple Consensus. Les méthodes classiques mathématiques coûtent beaucoup en ressources (processeur et mémoire vive) pour le traitement. Ransac, lui, se base sur les probabilités, plus exactement la probabilité de trouver dans un ensemble a priori incohérent et chaotique de données correspondantes à un motif ou à quelque chose de connu. C'est une méthode pour estimer les paramètres de certains modèles mathématiques. Pour notre application, l'algorithme permet de déterminer dans un nuage de points (le scan laser) LE meilleur motif présent dans le nuage. Nous l'utilisons donc pour trouver la meilleure droite sur notre scan. Ainsi, récursivement, nous obtenons toutes les droites de notre ensemble de points.

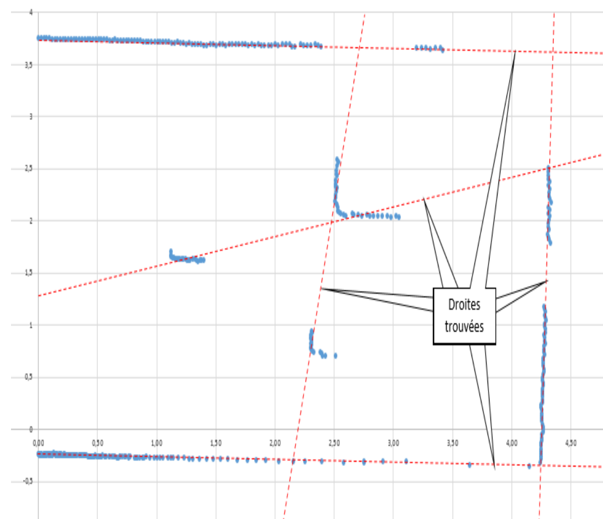
Cependant, les paramètres de cet algorithme doivent être choisis avec précaution :

- le nombre minimum de points pour dire que c'est bien une droite
- le nombre maximal d'itérations de l'algorithme
- une valeur seuil pour déterminer si un point correspond à un modèle de droite
- le nombre de données proches des valeurs nécessaires pour faire valoir que la droite trouvée est la meilleure des droites

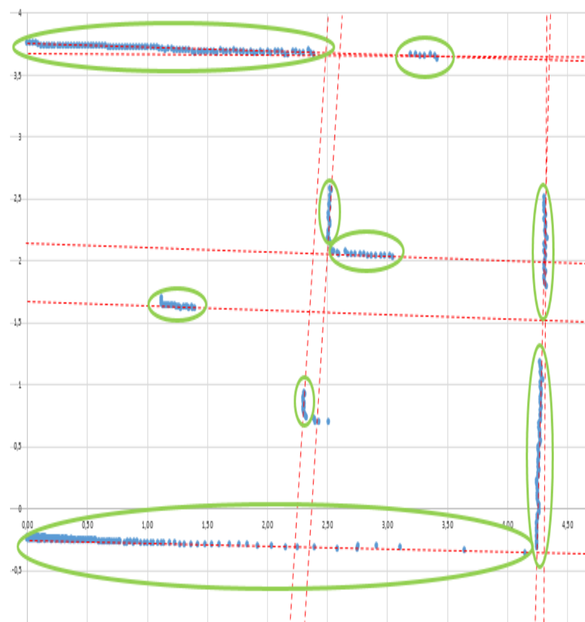
Sur un scan comme celui-ci :



On obtient une reconnaissance de droites avec nos paramètres optimaux comme celle-ci :



Si on pousse la recherche avec des paramètres plus fins on obtient en 10 fois plus de temps le résultat suivant :

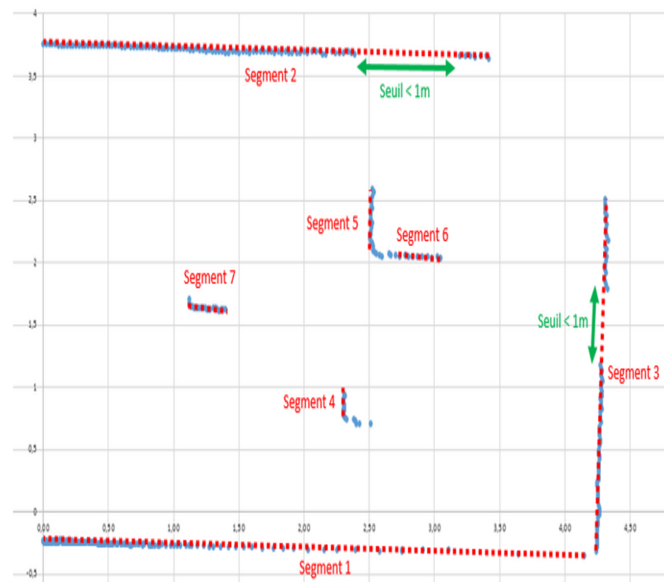


On voit donc toute l'efficacité de l'algorithme, donc voici ci-dessous le pseudocode :

```
tant que la liste contient assez de points
 itérateur := 0
 meilleur_modèle := aucun
 tant que itérateur < k
  on choisit deux points au hasard dans la liste
  on fabrique un modèle_possible à partir de ces deux points
  Pour chaque point de la liste qui ne sont pas dans le modèle_possible
   si le point s'ajuste au modèle_possible avec une erreur inférieure à t
    Ajouter ce point au modèle
   si le nombre de points est suffisant pour valider l'existence du modèle
    corrélation:= corrélation de la régression linéaire de ensemble_points
   si corrélation < corrélation du meilleur_modèle
    meilleur_modèle := modèle_possible
 stocker meilleur_modèle dans liste de modèles
 retirer meilleur_ensemble_points de la liste des points initiale
 recommencer avec la liste modifiée
```

c. *Extraction des machines*

Les droites ayant été détectées, l'étape suivante est de les transformer en segments, éléments sur lesquels on va ensuite travailler. Pour chaque droite, on parcourt l'ensemble de points correspondants. A chaque fois que l'on détecte un seuil important entre des points, on crée un nouveau segment. Après ça, on fait une régression linéaire sur tous les points du segment. On projette ensuite les points extrêmes sur cette régression. On réduit ainsi les effets de bord qui aurait pu perturber la mesure de la taille du segment. On s'assure aussi que les points extrêmes appartiennent vraiment au segment. On a donc le résultat suivant :



A partir de là, on filtre les segments de la bonne taille. Sur ce scan, on a un objet que l'on considérera comme une machine entre les segments 5 et 6. En réalité, et par la suite, nous utiliserons des scans avec des machines de la « vraie » taille (35*70). Donc, sur celui-ci, nous pouvons calculer les coordonnées d'un centre machine.

En pratique, on calcule 2 centres machines à partir des extrémités du segment, puis on choisit le centre le plus loin. En effet, l'un étant vertical et l'autre horizontal par rapport au laser, le moindre bruit du laser peut causer un basculement du centre machine d'un côté ou de l'autre du segment détecté mais logiquement, le centre d'une machine est derrière le segment détecté.

Au début, les machines mesurant officiellement 35 cm sur 70, on avait décidé de détecter à la fois les segments de 35 et de 70, mais le petit côté de la machine était en fait du plexiglas, qui laissait passer le laser… On a donc juste gardé la détection des grands côtés.

Machine
Segments détectables
Centre de la machines

Le tout est visible sur YouTube à l'adresse : https://www.youtube.com/watch?v=rKeqewpuLoQ

2) Correction de l'odométrie

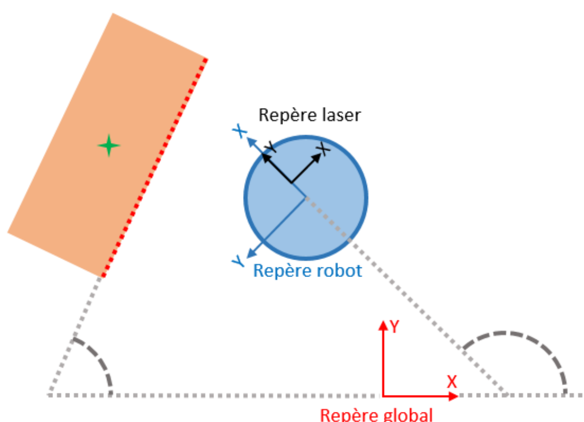
L'odométrie, la mesure de la position à l'aide des codeurs incrémentaux reliés aux moteurs, est censée être plus précise en ajoutant un gyroscope externe. Les nœuds ROS pour le robotino corrigent bien la position angulaire du robot en fonction des données du gyroscope. Ce qu'ils ne corrigent pas c'est la vitesse angulaire, il a donc fallu faire un nœud intermédiaire pour s'occuper de ce problème. Le travail de ce nœud consiste donc à calculer la différence d'angle entre la dernière publication et la nouvelle, et de diviser le tout par la période de fonctionnement du nœud. La seule difficulté est de travailler avec des angles en quaternion, mais une fonction de *tf* permet de convertir un angle en quaternion en angle en radian. Le calcul en ligne de code donne :

```
newOdom.twist.twist.angular.z = tf::getYaw(odom.pose.pose.orientation) - tf::getYaw(prec.pose.pose.orientation) /
                                (getTimeAsDouble(odom.header.stamp) - getTimeAsDouble(prec.header.stamp));
```

3) Extended Kalman Filter

a. *Data Association*

L'association de données sert essentiellement à transposer les machines vues au laser du repère laser au repère global. Avec l'odométrie correctement initialisée à la position réelle du robot dans le repère de la piste, que ce soit directement avec les mesures de l'odométrie réelle ou avec la position donnée par l'EKF, le changement de repère s'effectue de la même manière :

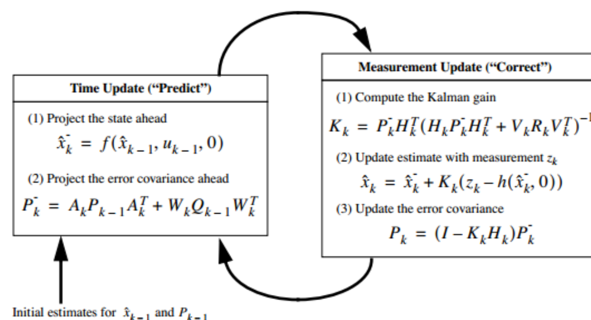


Le passage du repère laser au repère robot est une rotation et une translation de 10 cm, qui correspond à la position du laser sur le robot.
Le passage du repère robot au repère global est une rotation et une translation comprenant la position initiale ainsi que la mesure de l'odométrie OU de la position calculée par l'EKF.

b. *Algorithme de Kalman*

Pour faire de la localisation et de la cartographie simultanée, SLAM en anglais, il faut utiliser le maximum d'informations provenant des capteurs. Ici, nous utiliserons comme données les vitesses de commande ainsi que les données du laser. L'algorithme de Kalman, plus particulièrement le filtre de Kalman étendu, va nous permettre de fusionner les deux informations, tout cela avec un aspect filtrage qui va permettre de calculer l'erreur pour la compenser à chaque itération.

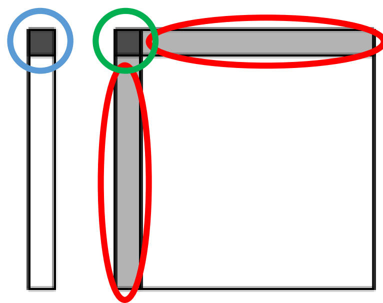
Le filtre fonctionne comme ci-dessous :



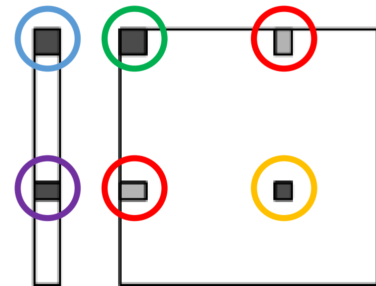
La partie *time update* prédit l'état du robot à l'itération suivante, elle utilise la vitesse du robot ainsi que la position précédente.

```
void EKF::prediction(){
  std::cout << "prediction" << std::endl;

  //calcul de la période pour la prédiction
  ros::Duration duree = ros::Time::now() - m_temps;
  double period = duree.toSec();

  //calcul de la position du robot pour l'instant n+1
  m_xPredicted(0) = m_xMean(0) + period*(cos(m_xMean(2))*m_cmdVel(0)-sin(m_xMean(2))*m_cmdVel(1));
  m_xPredicted(1) = m_xMean(1) + period*(sin(m_xMean(2))*m_cmdVel(0)+cos(m_xMean(2))*m_cmdVel(1));
  m_xPredicted(2) = m_xMean(2) + period*m_cmdVel(2);

  m_xMean.block(0,0,3,1) = m_xPredicted.block(0,0,3,1);

  MatrixXd Fx = MatrixXd::Identity(m_P.rows(),m_P.cols());
  Fx(0,2) =  m_cmdVel(0)*cos(m_xMean(2))*period;
  Fx(1,2) = -m_cmdVel(1)*sin(m_xMean(2))*period;

  //mise à jour de m_P
  m_P_prev = Fx*m_P*(Fx.transpose());
  //mise à jour du temps
  m_temps = ros::Time::now();
}
```

Le *m_xPredicted* sert de variable temporaire pour réaliser la prédiction, on met ensuite à jour le *m_xMean*, plus particulièrement la partie du vecteur d'état qui indique la position du robot. On met aussi à jour la matrice *m_P*. Cette matrice est le cœur du filtre. En effet, c'est un graphe sous forme matricielle de tous les éléments détectés et du robot. Initialement, cette matrice est vide, on l'initialise à la première prédiction. Ensuite, elle grandit au fur et à mesure des ajouts de machines. Ces dernières et le robot sont stockés en tant que x, y et theta dans le vecteur d'état et le lien entre tous est la matrice *m_P*.



Dans la partie prédiction on ne modifie qu'une partie des deux éléments clés :
Le haut du vecteur d'état correspond à la position du robot.
La partie en haut à gauche de la matrice correspond au vecteur déplacement du robot, et les premières lignes et colonnes en gris clair correspondent aux vecteurs positions entre chaque machine et le robot.



La partie *measurement update* corrige la position du robot et des machines vues à cet instant. A chaque observation d'un élément déjà présent dans le vecteur d'état, on calcule le gain de Kalman correspondant à la correction qu'il faut faire pour compenser le bruit mesuré. On corrige ainsi les parties ci-contre :

Dans le vecteur d'état, on corrige :

- la position du robot
- la position de la machine observée

Dans la matrice P, on calcule :

- le vecteur déplacement du robot
- la différence de position entre la machine précédemment observée et la machine actuellement observée
- la position de la machine par rapport au robot.

Le gain de Kalman n'est pas un gain global correspondant à l'ensemble des données sur les machines et le robot, c'est un gain calculé à chaque fois que l'on observe une machine. Il ne prend en compte que l'état précédent de l'objet et l'état actuel.

La difficulté dans le SLAM était d'ajouter au fur et à mesure les machines dans le filtre de Kalman. En effet, le filtre de Kalman est censé, dans son utilisation normale, corriger des positions déjà pré-remplies. Cependant, dans notre cas, c'était impossible puisque seulement les zones des machines étaient données aléatoirement par la Referee Box, l'arbitre de jeu, en début de partie.

L'ajout des machines perturbait beaucoup le filtre, nous avons donc ajouté de multiples conditions restrictives pour éviter les faux positifs. A chaque problème son test :

La zone de jeu étant partiellement ouverte, le laser peut « voir » des machines en dehors du terrain. Aussi, les murs mesurant environ 70 cm sont aussi considérés comme des machines.

➔ Chaque machine détectée est associée à sa zone si elle existe. Dans le cas contraire, elle n'est ni ajoutée ni corrigée.

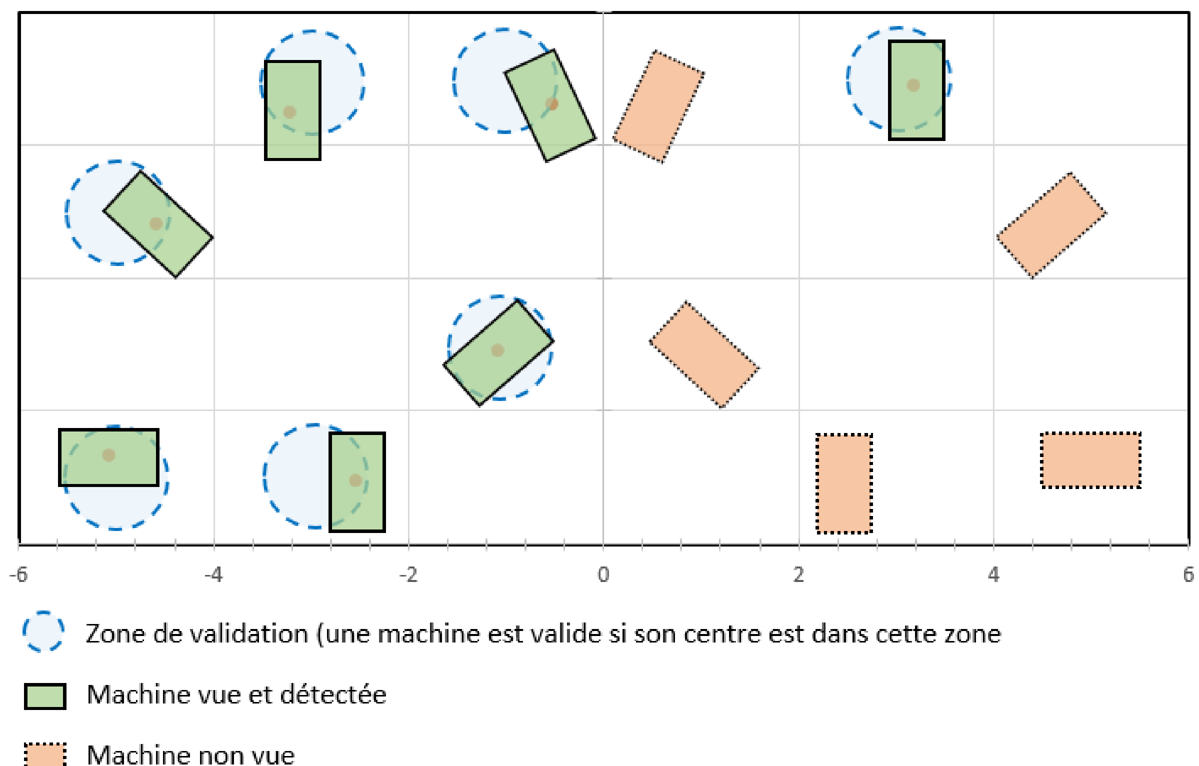
La discrétisation pendant la prédiction crée une dérive de la position du robot, donc un décalage du repère dans lequel sont faits les scans laser. On peut donc se retrouver avec des machines qui changent de zones.

➔ On limite les cas où les machines sont en bord de zone. Le centre de la machine doit donc se situer dans le cercle inscrit à la zone.

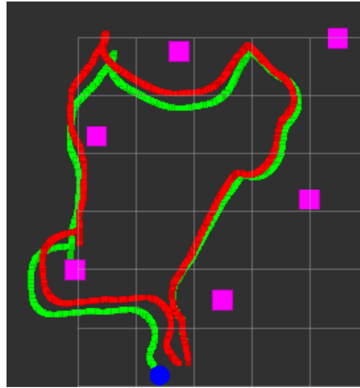
Dans le règlement, il doit toujours avoir assez de place entre les machines ou entre une machine et un mur pour qu'un Robotino puisse passer.

➔ On fait un test sur les machines les plus proches de la machine détectée, si la distance est supérieure à un seuil fixé, on la garde.

La détection de machine avec tous ces tests donne le résultat suivant :



On compare alors la trajectoire mesurée et la trajectoire calculée et corrigée avec l'EKF :



Une alternative a été mise en place dans le souci de pouvoir avoir une localisation fiable sans l'EKF, qui était non réglé. De nombreux tests ont été faits sur la piste officielle avec différentes configurations et nous ont permis de vérifier que l'odométrie ne dérivait que très peu. Il a donc été décidé de n'utiliser que l'odométrie et le scan laser pour la partie exploration.

En effet, chaque machine mesurée est envoyée à un nœud ROS qui va créer une carte et la compléter au fur et à mesure de la partie. Le robot n'est donc repéré rien qu'avec son odométrie. Le fait que la phase d'exploration (découverte du terrain) ne dure que 4 minutes nous a convaincu pour n'utiliser que cette mesure.

De plus, la carte créée sera utilisée pour la phase suivante, la phase de production. Cette phase consiste, au niveau de la localisation, à se repérer sur la piste et atteindre la machine demandée par l'exécuteur de tâches.

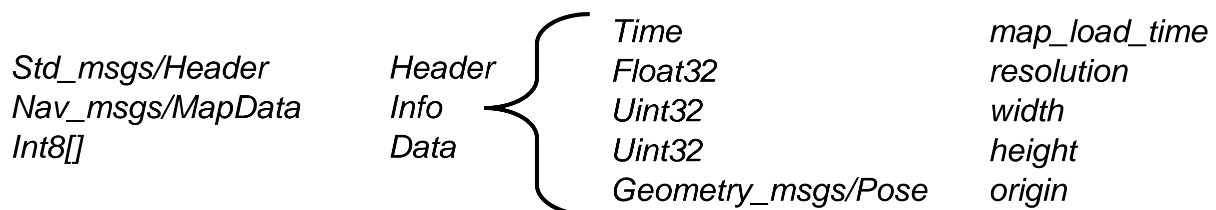
Pour cette phase qui dure 15 minutes, l'utilisation du filtre est nécessaire. Donc, avec les coordonnées des machines mesurées pendant la première phase, on crée le vecteur d'état et la matrice faisant le lien entre les machines. De là, la taille de ces deux éléments ne variant plus, on peut uniquement corriger la position du robot.

4) Grid Map

a. Génération de map vide

La map officielle mesure 12 m de longueur sur 6 m de largeur. Cependant, les zones de départ étaient en dehors de ces dimensions. Aussi, le règlement autorisait une sortie partielle du robot, donc, afin de couvrir correctement l'ensemble de la zone, nous avons choisi de générer une grille de 14m sur 8m.

Afin de générer cette grille, nous avons cherché un format qui nous permettait de définir la taille de la map mais aussi la résolution, l'origine et une donnée associée à chaque case.
Nous avons alors choisi de générer une OccupancyGrid dont voici les informations principales :



Toutes les infos dont nous avons besoin sont incluses dans ce type, pas besoin donc de réinventer la roue. En effet, le tableau de données *Data* nous permet d'associer à chaque case un pourcentage de possibilité de rencontrer un obstacle :

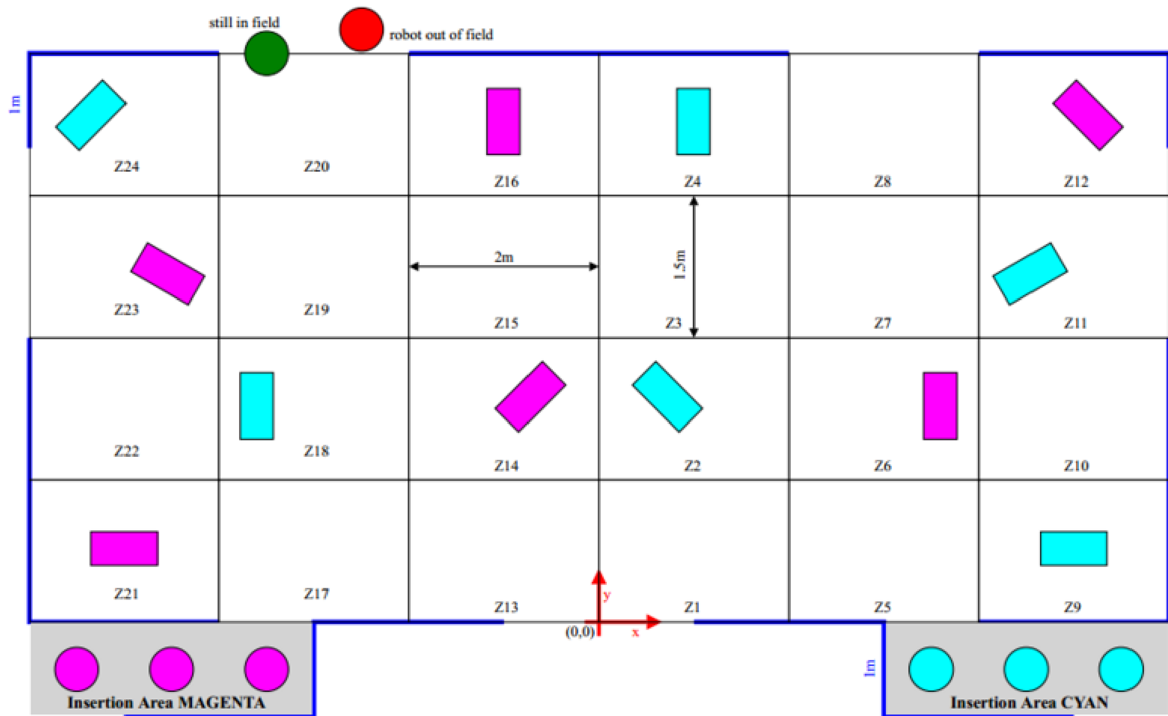
- 0 : pas d'obstacle
- entre 0 et 100 : dégradé sur la probabilité qu'il y ait un obstacle
- 100 : présence d'un obstacle

Pour l'origine de la piste, nous avons pris celui de la map officielle. Pour la résolution, nous en avons choisi une de 10 cm, ce qui est une taille à la fois suffisante comparé à la taille du robot et assez grande pour limiter les temps de calcul avec l'algorithme de recherche de chemin.

Il a été décidé avec le responsable de l'équipe que nous n'allions utiliser que 3 niveaux, sachant que la gradation pourra être utile pour la détection d'obstacles mobiles :

- Les murs à 100
- Les zones inaccessibles autour des murs à 50
- Les zones libres à 0

Ainsi la map officielle :

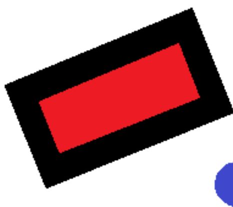


Devient la map vide visualisée grâce au logiciel de visualisation RVIZ :

b. *Prise en compte des obstacles fixes (machines)*

Au fur et à mesure de la cartographie réalisée dans le nœud EKF, les machines vues sont publiées en continu sur le topic */landmarks*. Il faut donc récupérer ces positions via une souscription au topic.

Le but est de remplir les rectangles correspondants à chaque machine en fonction des positions reçues.

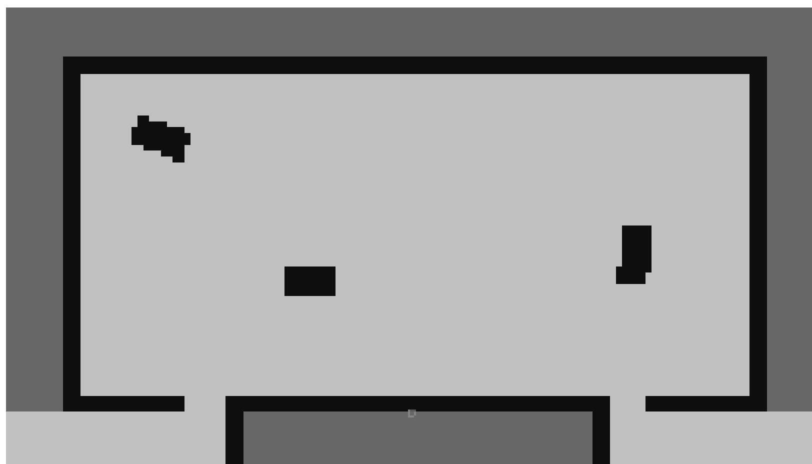


La taille d'une **machine** étant de 70 x 35 cm, et le **robot** mesurant 46 cm de diamètre, on ajoute une **distance de sûreté** de 25 cm (un peu plus que la moitié du robot) autour des machines pour que le robot ne percute jamais. Une machine est finalement un rectangle de 120 x 85.

Lorsqu'on reçoit une position machine, c'est en fait son centre et son orientation dans le cercle trigonométrique. Après avoir cherché sur internet des moyens de « pixelliser » un rectangle sur une grille, il est apparu que le centre n'était pas un moyen efficace pour dessiner la machine. On choisit donc un des coins du rectangle et plus précisément le coin en bas à droite de ce dernier.

On met alors tous les points de ce rectangle à une probabilité 100 dans le vecteur Map. Pour cela, on incrémente x et y à partir du coin, de sorte à longer le grand côté et de remplir chaque case (incrémentation de 5 cm). Une fois arrivé au bout, on incrémente x et y de 5cm sur la largeur à partir du coin et on recommence l'opération précédente jusqu'à remplir le complètement le rectangle.

Voici donc un résultat type obtenu avec 3 obstacles aléatoires sur la map:



Nous pouvons maintenant publier la Map et l'utiliser pour la recherche de chemin afin, au final, de naviguer à travers celle-ci.

5) Pathfinder

a. *Choix de l'algorithme*

La carte étant partiellement ou complétement créée, comment naviguer de manière optimale en trouvant le plus court chemin nous amenant devant une machine tout en évitant bien sûr les obstacles détectés ?

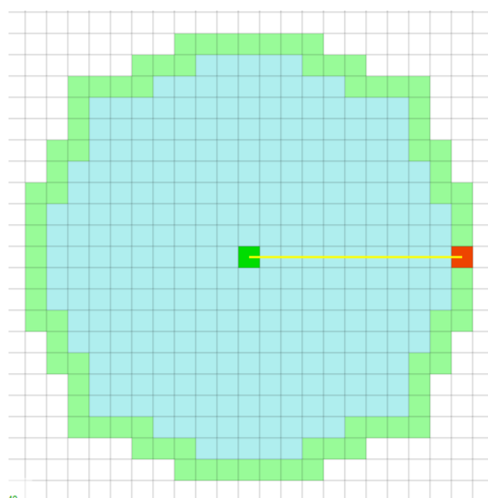
Deux algorithmes déterministes principaux sont utilisés pour la recherche du plus court chemin dans un graphe :

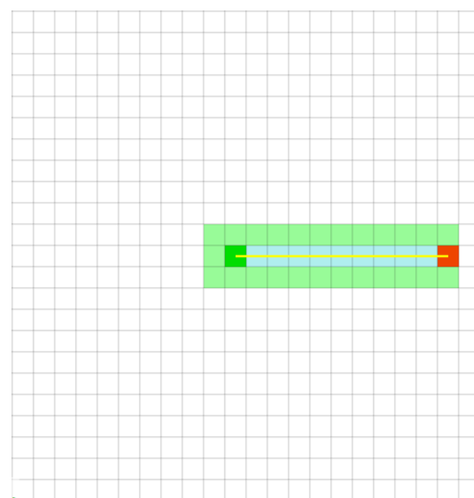
- L'algorithme de *Dijkstra*, qui permet de déterminer à coup sûr le chemin optimal. Il est, par exemple, utilisé pour le routage Internet. Cet algorithme cherche, à partir du point de départ et dans toutes les directions, le chemin le plus court pour arriver au point d'arrivée. Sur un graphe de la taille de notre map, le temps de calcul est trop important pour avoir un comportement temps réel correct.

- L'algorithme *A**, qui est beaucoup plus rapide à condition d'avoir une bonne heuristique. En pratique, c'est un bon compromis entre coût de calcul et optimalité de la solution. A la différence de Dijkstra, il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. C'est un algorithme simple, ne nécessitant pas de prétraitement, et ne consommant que peu de mémoire. L'heuristique est un élément significatif de cet algorithme et sera explicité par la suite.

Voyons la différence entre les deux pour un chemin très simple :

	Dijkstra	A*
temps	5,6349 ms	1.4548 ms
opérations	643	47





b. L'algorithme A*

Explications

L'algorithme A* est un algorithme itératif qui privilégie la recherche VERS le point d'arrivée. On va donc essayer de se rapprocher de la destination, en privilégiant les possibilités directement plus proches de la destination et en mettant de côté toutes les autres.

L'algorithme va donc d'abord se diriger vers les chemins les plus directs. Et si ces chemins n'aboutissent pas ou bien s'avèrent mauvais par la suite, il examinera les solutions mises de côté. C'est ce retour en arrière pour examiner les solutions mises de côté qui nous garantit que l'algorithme nous trouvera toujours une solution (si tant est qu'elle existe, bien sûr).

Notations

s	état initial
T	ensemble des états terminaux
h(x)	heuristique estimant le coût de x à T
k(x,y)	coût du passage de x à y, successeur de x
F(ermés)	ensemble des états développés (leurs successeurs ont été générés)
O(uverts)	ensemble des états générés mais non développés
g(x)	coût du trajet de s à x
f(x)	coût total estimé du trajet de s à T en passant par x
père(x)	état ayant généré x

Pseudo-Algorithme

```
O <- {s}; F <- Ø; g(s) <- 0; f(s) <- h(s);

Tant que O != Ø faire
  Extraire de O l'élément x tq f(x) est minimale;
  Insérer x dans F;
  Si x appartient à T alors
    Solution trouvée
  Sinon // Développement de x
    Pour tout y successeur de x faire
      Si y n'appartient pas à F U O ou g(y) > g(x) + k(x,y) alors
        g(y) <- g(x) + k(x,y)
        f(y) <- g(y) + h(y)
        père(y) <- x
        Insérer y dans O
      FinSi
    FinPour
  FinSi
FinTantQue
```

Heuristiques

- *Euclidienne (vol d'oiseau)*

```
float Map::heuristicEuclidean(Point const& pointDepart, Point const& pointDistant)
{
    float dist  = 0;
    float distX = 0, distY = 0;

    distX = fabs(pointDistant.getX() - pointDepart.getX());
    distY = fabs(pointDistant.getY() - pointDepart.getY());

    dist = std::sqrt(std::pow(distX,2)+std::pow(distY,2));

    return dist;
}
```

- *Manhattan (pas de diagonale)*

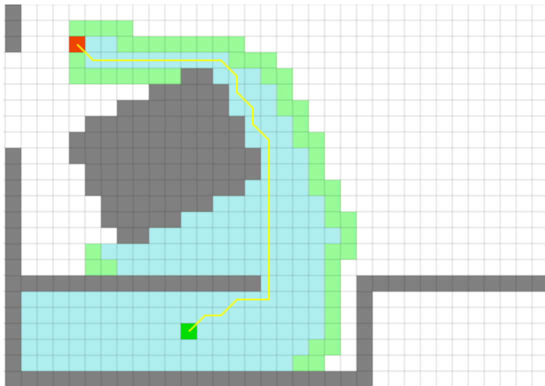
```
float Map::heuristicManhattan(Point const& pointDepart, Point const& pointDistant)
{
    float dist  = 0;
    float distX = 0, distY = 0;

    distX = fabs(pointDistant.getX() - pointDepart.getX());
    distY = fabs(pointDistant.getY() - pointDepart.getY());

    dist  = distX + distY;

    return dist;
}
```

- *Chebyshev (max de la distance selon x et de la distance selon y)*

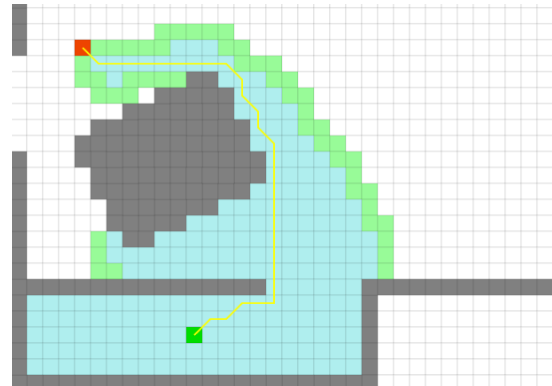
```
float Map::heuristicChebyshev(Point const& pointDepart, Point const& pointDistant)
{
    float dist = 0;
    float distX = 0, distY = 0;

    distX = fabs(pointDistant.getX() - pointDepart.getX());
    distY = fabs(pointDistant.getY() - pointDepart.getY());

    dist = std::max(distX,distY);

    return dist;
}
```

Exemples

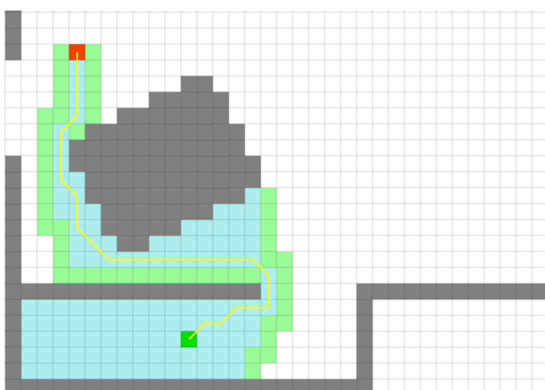
- *Euclidienne*



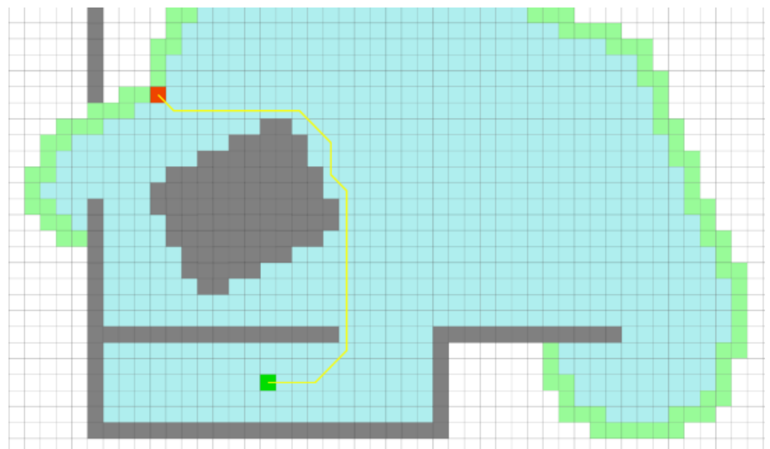
- *Chebyshev*



- *Manhattan*



- *Dijkstra*



	Dijkstra	A*		
		Euclidienne	Manhattan	Chebyshev
Temps ms	7,4361	3,3935	1,9838	4,0637
opérations	1646	432	298	499
longueur	31,49	31,49	32,9	31,49

On choisit donc l'heuristique Manhattan, qui permet un résultat correct (pas optimal cependant) en très peu d'opérations et de temps.

6) Path tracker

Ce nœud est censé suivre la trajectoire précédemment calculée en optimisant les déplacements. Cependant, cette partie n'ayant pu être finie pour l'Open German, nous ne pouvons pas en parler plus que ça. Malgré tout, un membre de l'équipe, Valentin Vergez, a codé une solution sur place à base de régulateur P.

Cette solution permet d'atteindre la position d'arrivée en passant par tous les points en séparant le déplacement angulaire du déplacement linéaire. Ainsi, on vise le point suivant et on avance.

Ainsi, on a l'asservissement suivant :

```
distance_avance = 0.1 (carreau de 10 cm)

si la longueur de tableau_pts_à_parcourir >=2 alors
    pt_suivant = tableau_pts_à_parcourir[1]
sinon
    pt_suivant = tableau_pts_à_parcourir[0]

dX = pt_suivant.x - pt_le_plus_proche.x
dY = pt_suivant.y - pt_le_plus_proche.y
angle = atan2(dY, dX)

pt_avance.x = pt_le_plus_proche.x + distance_avance*cos(angle)
pt_avance.y = pt_le_plus_proche.y + distance_avance*sin(angle)

# Rejoindre point d'avance
## Viser (en angle)
adj = pt_avance.x – position_actuelle.x
opp = pt_avance.y – position_actuelle.y
angle_réel = atan2(opp, adj)

erreur_angle = angle – angle_réel

vitesse_linéaire_selon_x  = 0.2
vitesse_linéaire_selon_y  = erreur_angle/10
vitesse_angulaire_selon_z = erreur_angle
```

Conclusion

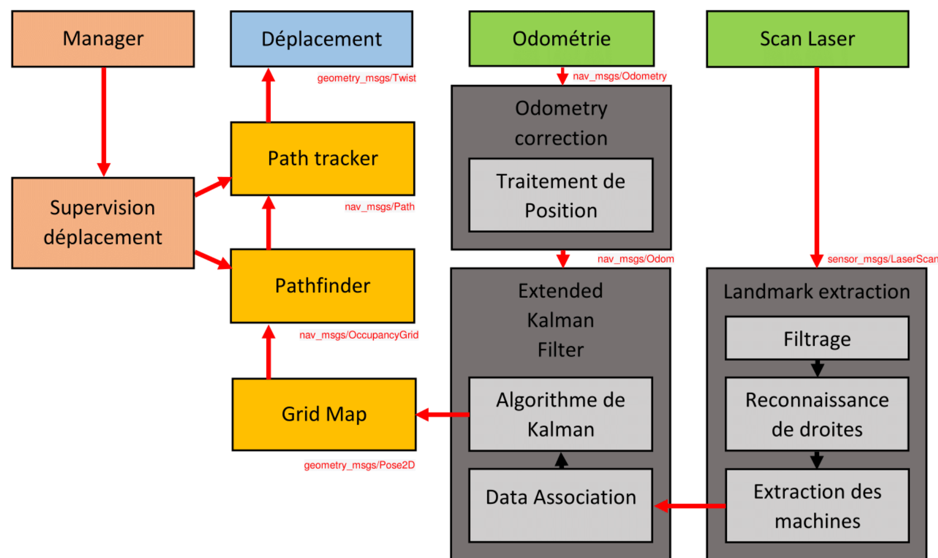
Cette expérience au sein d'une équipe d'étudiants, afin de mener à bien un projet de A à Z, a été vraiment enrichissante du point de vue technique, puisque nous avons abordé des algorithmes jusqu'alors inconnus, qui sont souvent utilisés en robotique mobile.

En effet, RANSAC, le filtre de Kalman ou l'algorithme A*, sont des algorithmes difficiles à comprendre au premier abord mais sont utilisés dans tout ce qui touche à la navigation. Ainsi, la compréhension de ces outils nous a permis d'envisager des améliorations concernant à la fois le fonctionnement et l'implémentation de nos programmes. Le travail fourni jusqu'alors servira de base pour les années suivantes.

Du point de vue gestion de projet, le programme et les différentes deadlines avaient été donnés par notre encadrant, Vincent Coelen. Son expérience dans ce domaine nous a vraiment boosté et aidé dans la réalisation de ce projet et de sa conclusion : notre 2ème place à l'Open German.

Nos résultats ont encouragé l'équipe à continuer, la RoboCup en Chine en juillet puis l'Open en France l'année prochaine ainsi que la RoboCup à Leipzig.

Annexe



Code concernant la partie ***Landmarks extraction*** :

https://github.com/PyroTeam/robocup-pkg/blob/devel/navigation/localisation/src/landmarks_detection_utils.cpp

et

https://github.com/PyroTeam/robocup-pkg/blob/devel/navigation/localisation/src/landmarks_extraction_node.cpp

Code concernant la partie ***Odometry correction*** :

https://github.com/PyroTeam/robocup-pkg/blob/devel/navigation/localisation/src/odometry_correction_node.cpp

Code concernant la partie ***Extended Kalman Filter*** :

https://github.com/PyroTeam/robocup-pkg/blob/devel/navigation/localisation/src/EKF_class.cpp

et

https://github.com/PyroTeam/robocup-pkg/blob/devel/navigation/localisation/src/EKF.cpp