Rapport

Projet ANDROIDE

Equipe :

Jérémy Dufourmantelle
Ethan Abitbol
Elias Bendjaballah
Jules Cassan

Encadrant :

Jean-Michel Ilie
Farouk Meddah

sujet : Amélioration de la conduite d'un robot roulant intelligent et autonome

Github : https://github.com/jdufou1/P-ANDROIDE

Table des matières

1 Introduction

Deux chercheurs du LIP6 (laboratoire d'informatique paris 6 - Jussieu), Jean-Michel Ilié et Farrouk Meddah souhaitent développer des objets intelligents autonomes, au sens où ils auront la capacité de repousser à leur maximum l'intervention humaine. Il existe déjà une version de ce robot développée dans le cadre du projet E-HoA [1]. Dans ce projet, nous cherchons à **améliorer la conduite/les mouvements d'un véhicule autonome dans un espace routier**, en particulier dans sa prise de virages. L'objet considéré dans ce projet est un robot roulant, au nom de TurtleBot3, représentant un véhicule terrestre intelligent sur lequel s'appuiera un acteur humain pour mener à bien ses missions, comme livrer des médicaments à des pharmacies réparties dans une ville dont on connaît la topologie. Pour répondre a ce problème il a été décidé que le circuit et l'ensemble des virages qui le compose seront représentés par une suite de **courbes de Bézier** et plus précisément **des courbes de Bézier quadratiques**, qui est un outil flexible et facile de représentation. En effet, définis à partir de 3 points, une courbe de bézier décrit assez bien l'allure d'un virage et peut par la suite être subdivisée pour une représentation plus précise ou adaptée à la situation. Donc, en exploitant des informations géométriques concernant la sinuosité de la route, on pourrait améliorer la technique de déplacement du robot roulant sur un espace routier.

Comment améliorer la conduite d'un robot roulant intelligent et autonome par la méthode des courbes de Bézier ?

Tout d'abord nous allons commencer par présenter qu'est ce que les courbes de Bézier pour ensuite faire une étude sur la faisabilité d'une approche déterministe de la conduite autonome du véhicule avec des routes représentés par ses courbes de Bézier. Les algorithmes implémentés vont devoir être testés et vérifiés sur des exemples de circuits donc un simulateur propre au projet a été implémenté afin de démontrer les calculs géométriques des courbes de bézier et de vitesse mais aussi de simuler les comportements de notre robot.
Parallèlement a cela, nous avons aussi générés des routes de manière automatisé sur le simulateur GAZEBO en 3D , qui fera l'intermédiaire entre l'implémentations des algorithmes et les tests sur robot réels afin d'avoir un environnement sécurisé, des sensations proches de la conduite automobile et d'émuler les conditions réelles.

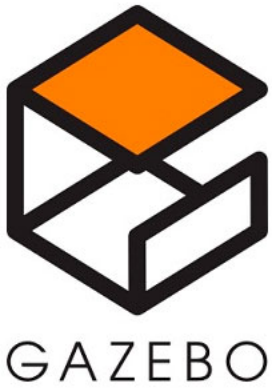
Le lien vers le dépot git du projet :

```
git clone https://github.com/jdufou1/P-ANDROIDE
```

2 État de l'art

Tout d'abord, l'étape 1 de notre projet fut d'acquérir les rapports de projets, de stages des années précédentes afin de pouvoir assimiler dans tout ses aspects le projet concerné. Pour cela, on a regroupé les éléments importants ci dessous.

2.1 Gazebo

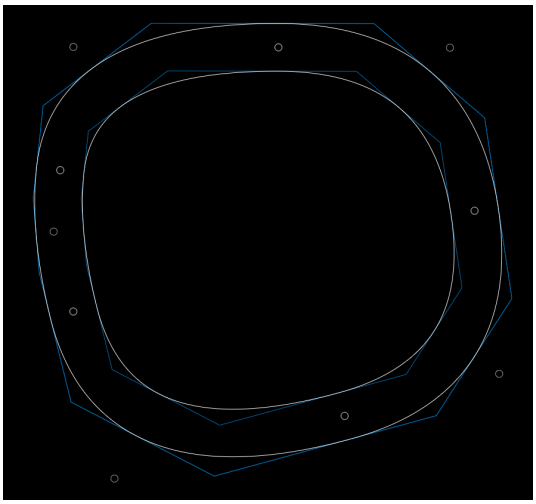


Le framework Gazebo consiste en une série d'outils permettant de simuler un environnement tridimensionnel avec des caractéristiques physiques intégrées. Gazebo se base sur le moteur 3D open source et multi-plateforme OGRE pour construire un rendu sur la base d'un octree définissant l'environnement courant ainsi que différentes physiques sous forme d'ODE ou de Bullet afin de détécter les collisions entre objets, simuler la gravité, la friction etc ... Le point fort de Gazebo est cependant au niveau de sa capacité à simuler et lire des données de capteurs de nature très diverses : cameras, lasers, lidars, radars ... grâce à des logiciels de la même famille tels que rViz.
Au démarrage, Gazebo charge un fichier SDF qui définit le monde (world) de la simulation étudiée. Le format SDF est un format open source basé fortement sur la syntaxe XML et définissant les objets présents dans la simulation ainsi que leurs caractéristiques. Nous utiliserons ces caractéristiques et d'autres spécificités de Gazebo pour porter nos travaux dans un environnement 3D et créer nos propres environnements de simulation.

2.2 Road Map Editor

Road Map Editor est une application Web développé essentiellement en Javascript, HTML et CSS par Nell Flaharty, ancien étudiant en M1 SAR [2]. Cet outil permet à un utilisateur de pouvoir créer des circuits grâce à des courbes de Bézier de manière interactive avec une interface graphique.

Un exemple de circuit par courbe de bézier crée par Road Map Editor :



3 Courbes de béziers

Inventés en 1962 par l'ingénieur français Pierre Bézier pour Renault qui les utilisa pour concevoir des pièces automobiles à l'aide d'ordinateurs, les courbes de béziers sont des courbes polynomiales paramétriques. On souhaite construire des courbes lisses à partir d'un nombre fini de points de contrôles.
Une courbe de Bézier est définie par un ensemble de points de contrôle P0 à Pn, où n est appelé l'ordre de la courbe (n = 1 pour linéaire, 2 pour quadratique, etc.). Le premier et le dernier point de contrôle sont toujours les extrémités de la courbe. Cependant, les points de contrôle intermédiaires (le cas échéant) ne se trouvent généralement pas sur la courbe.

Pour mieux comprendre comment cela marche, prenons un exemple :
Soit 3 points de contrôle P_0, P_1 et P_2. Notre but est de tracer une courbe allant de P_0 à P_1 tel que la tangente en P_0 soit $\overrightarrow{P_0P_1}$ et la tangente en P_2 soit $\overrightarrow{P_1P_2}$.

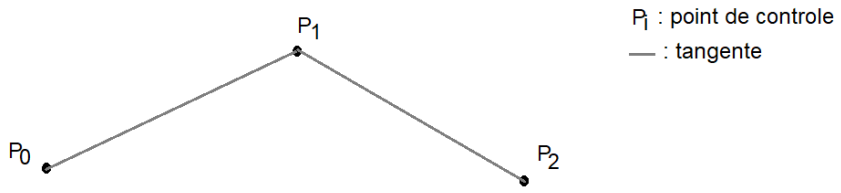


FIGURE 1 – Exemple de 3 points de contrôle avec le tracé des deux tangentes initiales

L'idée de Pierre Bézier est de procéder par barycentrage successif [3]. On trace le segment entre le milieu du segment $[P_0, P_1]$ et le milieu du segment $[P_1, P_2]$ et ainsi de suite.. On obtient finalement :

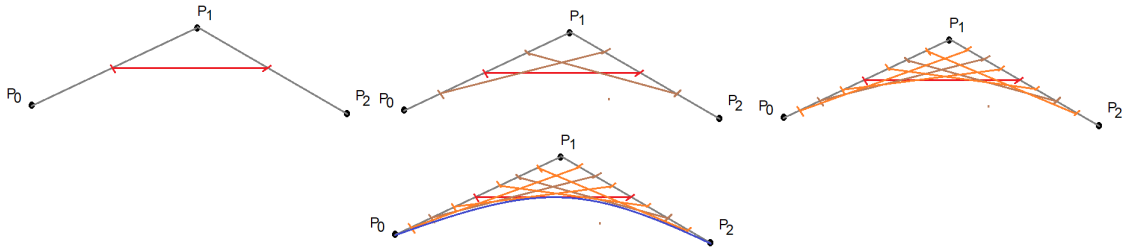


FIGURE 2 – Déroullement de la méthode par barycentrage successif

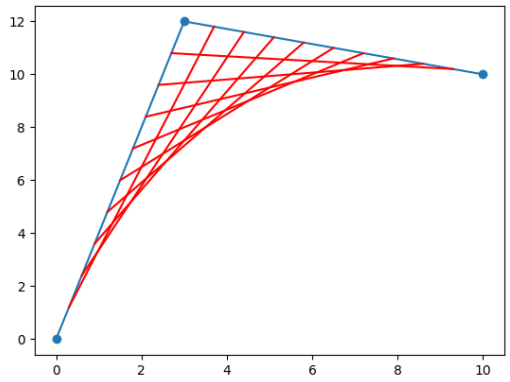


FIGURE 3 – Courbe de bézier obtenu avec 3 points de contrôle.

Ce qu'on a fait avec 3 points peut aussi être réalisé avec quatre points, c'est ce qu'on appelle : 'les courbes de Bézier cubique'

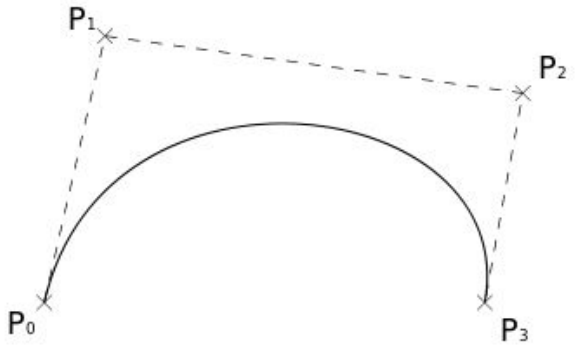


FIGURE 4 – Exemple d'une courbe de bézier cubique.

Comme mentionné plus haut, tout dépend de l'ensemble de points de contrôle choisis, dans notre cas nous allons nous intéresser aux **courbes de Bézier quadratiques**, 3 points de contrôles pour une raison analytique.

3.1 Les courbes de Bézier quadratiques avec subdivision sélective

Une courbe de bézier quadratique est donc défini par 3 points de contrôle P_1, P_C et P_2. Nous allons utiliser une méthode basé sur la subdivision. C'est à dire que au départ nous allons subdivisé la courbe de bézier afin de générer une approximation d'offset avec d'autres segments quadratiques de Béziers [4].

La courbe commence a P_1 et fini a P_2. Notons P_t le point le plus proche de la courbe a P_C :

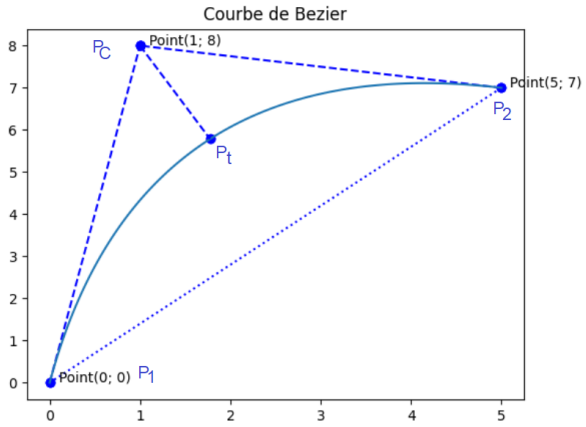


FIGURE 5 – Représentation de la courbe de bézier quadratique et du point P_t

De P_t, on obtient de nouveaux points de contrôle et de nouveaux P'_t :

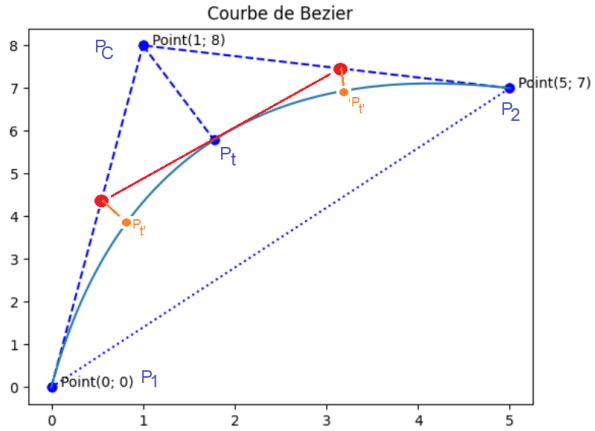


FIGURE 6 – Mise à jour de la figure précédente par ajout de la tangente au point P_t

Et ainsi de suite... On trace après les points de décalages Q_{ti} à une distance perpendiculaire de la direction de chaque point.

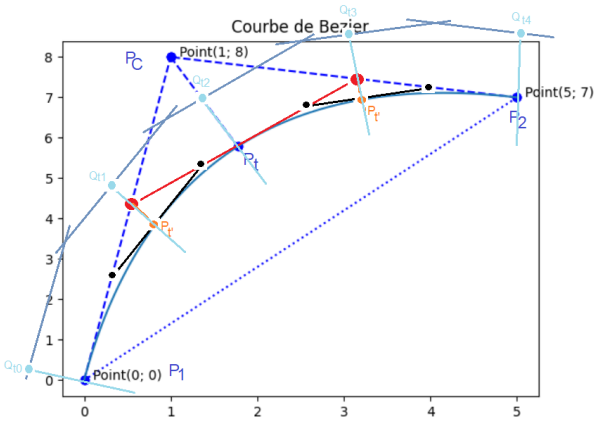


FIGURE 7 – Déroulement de la méthode par subdivision sélective et représentation des points Q_i

On peut voir que deux droites issus des points $Q_t i$ et $Q_t i + 1$ se croisent en un point qui seront les points de contrôle de chaque sous segment de Bézier pour le décalage. Enfin, on peut dessiner le chemin d'approximation du décalage en cousant des segments de Bézier quadratiques.

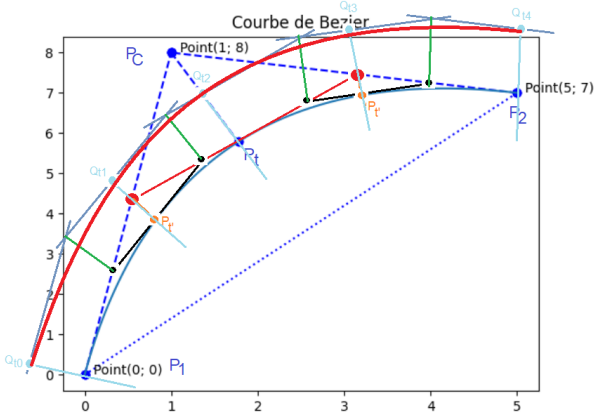


FIGURE 8 – Dessin du chemin d'approximation du décalage

Le processus est le même pour les deux cotés de la courbe. Grâce à cela on peut tracer des routes basé sur les courbes de Bézier quadratique utilisant trois points de contrôles. Cette algorithme est efficace car il permet la création rapide de la courbe en effectuant des divisions par 2. Il est de plus non récursive donc il est exécuté le

nombre de fois que la subdivision de Bézier quadratique se produira. La subdivision minimale de la courbe est un (divisé uniquement à Pt).

4 Étude de faisabilité d'une approche déterministe de la conduite autonome du véhicule

Afin de conduire de manière autonome, nous proposons une approche déterministe basé sur la connaissance de la route sous forme de courbe quadratique de Bézier avec trois points de contrôle (dont les deux points d'extrémités). L'appareil autonome devra alors avoir conscience de **sa position** à tout instant ainsi que dans un premier temps **une vue globale** de la route. En fonction de ces informations, le décisionnaire devrait être capable de conduire de manière sûr sur un circuit (donc on supposera qu'il n'y a aucune intersection).

4.1 Représentation de la connaissance de la route

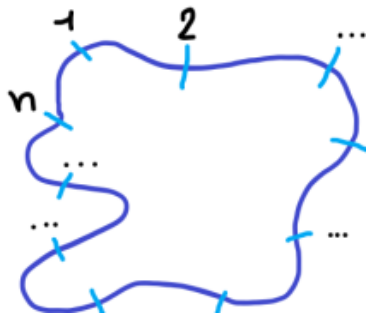


FIGURE 9 – Exemple de partition d'un circuit

Pour représenter la structure de la route en mémoire, nous avons imaginé de partitionner en N triplets de points où chaque triplet constitue une portion de la route sous forme de courbe quadratique de bézier. Nous noterons cette liste $\mathbf{L}$ tel que :

$$\mathbf{L} = [(P_1, P_C, P_2)_1, ..., (P_1, P_C, P_2)_N]$$

Il sera alors facile pour nous à partir d'un triplet $(P_1, P_C, P_2)_i, \forall i \in \{1, ..., N\}$ de calculer la sinusoité de la route c'est à dire son angle, le nombre de points critique et enfin sa difficulté sur une échelle. Ainsi lorsque le robot à cette représentation du

circuit en mémoire, il peut alors passer à l'étape de prise de décision de vitesse à une position donnée et une vitesse donnée.

Pour connaître la position du point B_i, le prochain point de contrôle sur la route, sachant notre position courante M, le décisionnaire pourrait utiliser deux solutions. La première consiste à enregistrer dans un buffer (cache) les portions de route déjà passées jusqu'à $(P_1, P_C, P_2)_k$ et considérer le triplé $(P_1, P_C, P_2)_{k+1}$. Ou bien, pour économiser de la mémoire et utiliser du temps de calcul à la place, on pourrait appliquer une procédure notée **P** qui va filtrer la liste de manière à placer le triplé correspondant à la portion de route dans laquelle il se trouve en tête de liste.

Algorithm 1 P : Mise à jour de **L**

Require: M : position actuelle, **L** : liste de triplet
  $dist_{min} \leftarrow 0$
  $index \leftarrow 0$
  for i in range(|(**L**)|) **do**
    $P_1, P_C, P_2 \leftarrow \mathbf{L}[i]$
    $\Delta \leftarrow |P_1 - M| + |P_2 - M|$ (metrique que l'on peu changer)
    if $\Delta > dist_{min}$ **then**
      $dist_{min} \leftarrow \Delta$
      $index \leftarrow i$
    end if
  end for
  $\mathbf{L} \leftarrow \mathbf{L}[i :]$

Il doit être appelé à chaque fois que le décideur veut actualiser sa base de connaissance dans le but de prendre une décision sur la vitesse à appliquer par exemple. En ce qui concerne la complexité de cette algorithme, on fait que des affectations et calcul en O(1) ainsi que une boucle sur la taille de L donc cet algorithme à une complexité linéaire en $O(|(L)|)$.

4.2 Phase de calcul de la vitesse à prendre à un instant t donné

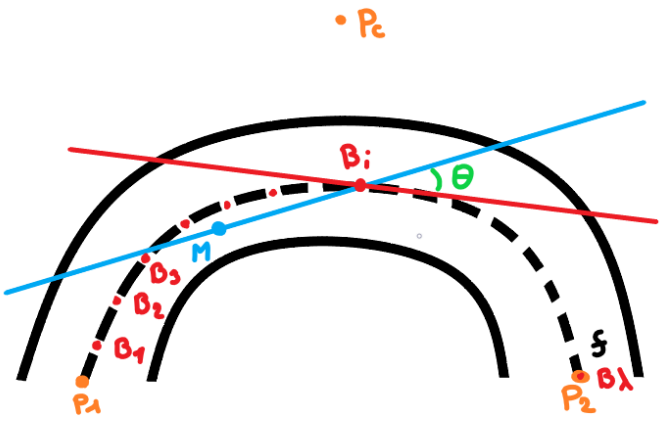


FIGURE 10 – Représentation d'un circuit avec les points B, la tangente au point B_i, la droite [M,B_i] et l'angle θ

Dans cette partie, nous allons étudier une approche pour calculer la vitesse linéaire et la vitesse angulaire maximum dans un virage en fonction de la position actuelle M, d'un point destination B_i placé sur le milieu de la route et d'une fonction de courbe de Bézier f pour la route. Pour obtenir le point B_i, nous proposons l'algorithme suivant :

Algorithm 2 FoundTheBestB_i

Require: M : position actuelle, **L** : liste de triplet, ϵ : pas de découpage de la route, f : fonction de la courbe de bézier
$B_1, ..., B_\lambda \leftarrow f(0), ..., f(k.\epsilon), ..., f(1)$ où $\forall k \in \mathbf{N}, 0 < k.\epsilon < 1$
$BestB_i \leftarrow B_1$
for i in range(2,λ) **do**
  if MB_i ne coupe pas la route **then**
    $BestB_i \leftarrow B_i$
  end if
end for
return $BestB_i$

De cette manière, l'algorithme nous permet de trouver le point B_i idéal le plus éloigné possible pour pouvoir anticiper le plus tôt possible la vitesse à adopter dans le virage en question. Nous pouvons aussi remarquer que si le virage est une ligne droite alors le point B_i retourné par l'algorithme sera le point $f(1)$ qui sera situé sur la même droite que notre position M.

Nous pouvons alors définir la vitesse linéaire maximum v_{lmax} autorisée en mètre/-seconde pour que notre robot fasse le trajet du point M au point B_i de la manière suivante :

$$tan(\theta) = \frac{V^2}{R*g}$$

$$V = \sqrt{tan(\theta) * R * g}$$

Ou V est la vitesse linéaire, θ l'angle du virage , R le rayon du virage et g la constante de l'accélération de la pesanteur ($g = 9.81m/s^2$). En ce qui concerne la complexité de cet algorithme, on commence par affecter à chaque B une valeur entre 0 et 1 par la fonction de la courbe de bézier en 0(1). Ensuite, on fait une boucle for de taille λ dans lequel on fait un if en O(1) et enfin on renvoie le meilleur B_i. Donc la complexité global de cet algorithme est en O(λ).

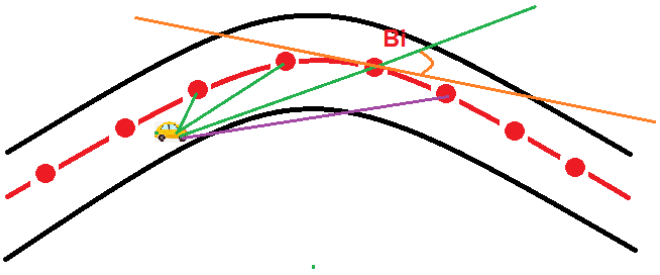


FIGURE 11 – Choix du meilleur point B

4.2.1 Calcul de l'angle du virage θ

Nous allons alors chercher à exprimer θ en fonction de données que nous connaissons, c'est à dire la fonction de courbe de Bézier f et les points M et B_i.

Pour cela, nous définissons l'équation de la droite (MB_i) noté $d_1 : a_1x + b_{i1}$
Coefficient directeur de la droite :

$$a_1 = \frac{y_{Bi} - y_M}{x_{Bi} - x_M}$$

Nous savons aussi que la droite passe par le point M, donc :

$$y_M = a_1 x_M + b_{i1}$$

$$b_{i1} = y_M - a_1 x_M$$

Ce qui donne :

$$d_1 : \frac{x_{Bi} - x_M}{y_{Bi} - y_M} x + (y_M - \frac{x_{Bi} - x_M}{y_{Bi} - y_M} x_M)$$

Nous allons maintenant chercher à calculer la droite tangente au point d'abscisse de B_i sur la fonction f.

Pour calculer cette droite, nous allons calculer deux points de cette tangente grâce à la fonction *lerp*.

$$lerp(P_1, P_2, t) = (1 - t).P_1 + t.P_2$$

Cette fonction a d'excellentes propriétés car elle nous permet d'obtenir deux points P_{1c} et P_{2c} qui appartiennent à la tangente tel que :

$$P_{1c} = lerp(P_1, P_C, t)$$

$$P_{2c} = lerp(P_C, P_2, t)$$

Avec t, que l'on a eu début lorsqu'on cherche le B_i grâce à l'algorithme FoundTheBestB_i . Ainsi, nous obtenons la droite d_2 de la même manière que la droite d_1 en calculant le coefficient directeur et l'ordonnée à l'origine à l'aide des points P_{1c} et P_{2c}.

Enfin, nous savons d'après la formule que l'angle formé entre deux droites d_1 et d_2 est :

$$\alpha = \arctan(\tfrac{a_2 - a_1}{1 + a_1 a_2})$$

4.2.2 Calcul de la vitesse angulaire v_a

On propose ici une méthode permettant de calculer la vitesse angulaire en fonction de la vitesse linéaire v_l, de la distance entre le point M et B_i.

$$v_l = \frac{D_{MB_i}}{t_{rotation}}$$

Ici, on va exprimer la distance MB_i en mètre avec la distance euclidienne dans un plan $2D$ normé en mètre. Ce qui nous donne :

$$d_{MB_i} = \sqrt{(x_{Bi} - x_M)^2 + (y_{Bi} - y_M)^2}$$

Il faut alors que l'on calcul le temps de rotation du robot en fonction d'un angle θ qui va être exprimé en seconde.

$$t_{rotation} = \frac{\theta}{v_a}$$

Nous remplaçons donc les différents termes pour obtenir la formule de vitesse linéaire maximum suivante :

$$v_a = \frac{\arctan|\frac{a_2 - a_1}{1 + [a_1 * a_2]}| * v_l}{\sqrt{(x_{Bi} - x_M)^2 + (y_{Bi} - y_M)^2}}$$

4.2.3 Test sur les formules proposées

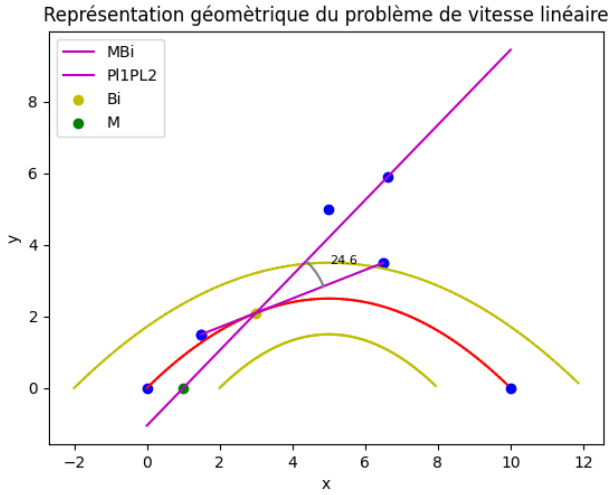


FIGURE 12 – Représentation géométrique du problème de vitesse linéaire

Cette figure illustre le problème du calcul de vitesse linéaire de manière géométrique en utilisant les formules que l'on présenté ci-dessus. Sur cette représentation le point M est situé en $(1, 0)$ P_1 en $(0, 0)$, P_C en $(5, 5)$, P_2 en $(10, 0)$ et enfin B_i en $(3, 2)$

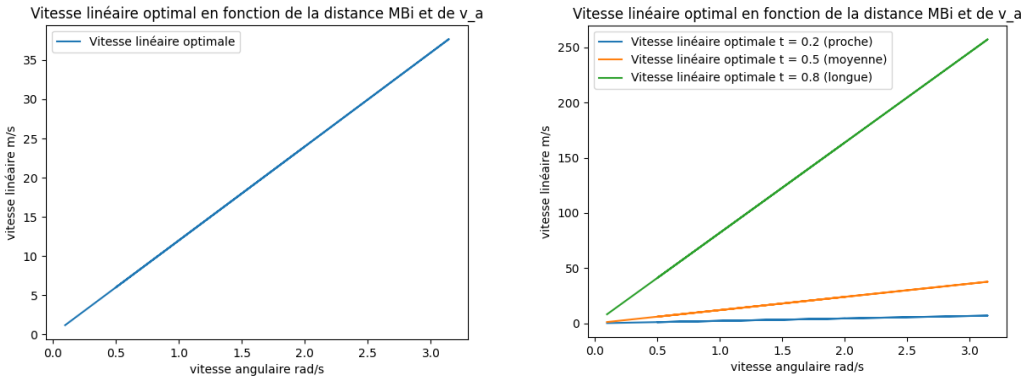


FIGURE 13 – Vitesse linéaire en fonction de la distance MB_i et de v_a

Nous voyons que sur cette figure, plus la distance entre M et Bi est petite, plus la

vitesse optimale est faible. Nous voyons aussi que plus la vitesse angulaire augmente, plus la vitesse linéaire augmente.

4.2.4 Choix du partitionnement d'une route

Dans cette partie, nous allons nous intéressé à fixer l'hyper paramètre ϵ que nous avons vu apparaître dans l'algorithme FoundTheBestB_i . En effet, il peut être judicieux de faire varier ϵ, donc le nombre de partitionnement de notre route en fonction de la sinuosité du virage et de sa difficulté. En effet, plus le virage sera difficile, plus il sera nécessaire d'augmenter ϵ afin d'augmenter le partitionnement du virage et de pouvoir mieux aborder un virage compliqué. Cependant, au contraire, si nous sommes devant une route sans virage, il serait préférable de ne presque pas partitionner la route afin d'économiser du temps de calcul.

4.3 Définition de l'algorithme général déterministe par méthode de Bézier

4.3.1 Pseudo-code

Voici ci-dessous le pseudo-code de l'algorithme général déterministe par méthode de Bézier que nous appelons aussi algorithme basé sur les courbes de Bézier. Cet algorithme reprends les étapes de calculs sur les courbes de Bézier, de vitesse linéaire et angulaire ainsi que les calculs de changement de direction.

Algorithm 3 Algorithme Général Déterministe par méthode de Bezier

Require: M : position actuelle, **L** : liste de triplet

1: **while** Véhicule actif **do**
2: $B_i \leftarrow$ algorithme de recherche de point idéal (FoundTheBestB_i)
3: **if** B_i visible par le robot **then**
4: $\alpha \leftarrow$ Calculer l'angle de correction de la trajectoire en fonction de l'angle formée par le vecteur de direction et la droite (MB_i)
5: Effectuer une rotation avec l'algorithme de rotation (Procédure de réajustement de la direction)
6: Adapter la vitesse linéaire et angulaire en fonction de α
7: **else**
8: Ne pas changer le vecteur directeur
9: Augmenter la vitesse linéaire
10: Diminuer la vitesse angulaire
11: **end if**
12: **end while**

En ce qui concerne la complexité de cet algorithme :
Tout se passe dans le while qui correspond a la simulation, tant que nous restons dans la simulation le while renvoie True. Donc on va se focaliser sur ce qui se passe à l'intérieur du While. Tout d'abord, on calcul le point B_i idéal grâce à l'algorithme (FoundTheBestB_i) qui a une complexité de O(λ). Ensuite on teste la condition en O(1), dans le pire des cas nous rentrons dans le if et donc nous calculons le α en O(1) puis effectue l'algorithme de la Procédure de réajustement de la direction qui est négligeable. Donc par tour de boucle on a une complexité en O(λ) et une complexité global : O(temps(véhicule_actif)*λ).

Toutefois, si le robot est placé initialement sur une position en dehors de la route, le robot ne pourra pas atteindre un point B_i et roulera en ligne droite.

4.3.2 Implémentation et test de l'algorithme général déterministe par méthode de Bézier

Dans cette partie nous allons présenter des mesures que nous avons réaliser avec différents graphes où on mesure le temps par tour du robot sur un circuit donné.

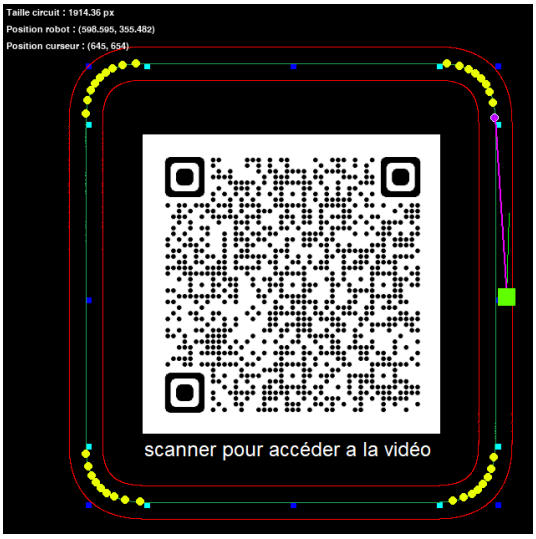


FIGURE 14 – Environnement de test pour analyser les algorithmes

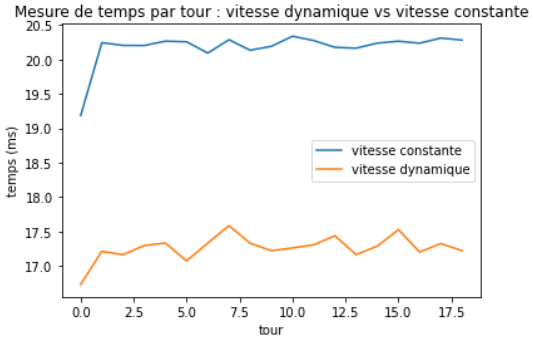


FIGURE 15 – Mesure de temps par tour : vitesse dynamique vs vitesse constante

Voici une mesure, qui a été réalisé sur un circuit de taille 1914.16 pixels. Nous avons fait tourner sur 20 tour pour analyser le temps mit par l'algorithme générale déterministe par méthode Bézier et un algorithme fonctionnant similairement mais avec une vitesse linéaire constante (de 8 $pixel/s$) qui correspond à la vitesse maximale pouvant être pris par le robot sur tous le circuit dans le but de ne pas sortir de la route.

Nous voyons sur cette figure que l'algorithme suivant une vitesse dynamique nous donne des résultats plus satisfaisant que l'algorithme suivant une vitesse constante.

Des résultats que nous nous attendions à obtenir en raison de l'augmentation de la vitesse dans les zones peu incurvées.

De plus, l'algorithme générale déterministe par vitesse dynamique prends en compte une constante d'accélération linéaire de 1.1 m/s^2 et une constante de décélération de linéaire de 3 m/s^2.

Les deux méthodes ont permis au robot de ne pas sortir de la route pendant tous les tours. Enfin, vous pourrez retrouver des démonstrations et des explications à propos du simulateur en fin de rapport.

4.3.3 Limites de l'algorithme

D'après nos différents tests, nous avons du ajuster à la main la constante d'accélération linéaire du véhicule car certaines fois, lors d'un virage trop serré, le robot sort du virage. Pour régler ce problème il serait pertinent d'utiliser des méthodes par apprentissage pour apprendre ces valeurs critiques de constante d'accélération.

De plus, notre algorithme ne prend pas en compte la position initiale du véhicule ce qui pose problème au moment du calcul du point B_i dans l'algorithme car il se sert de ce point pour donner une direction au véhicule. Il y a aussi beaucoup de paramètre à fixer comme la taille de découpage de notre circuit, le seuil de changement de direction, le choix des constantes d'accélération etc...

5 Conception d'un simulateur 2D

Dans cette partie, nous allons présenter le simulateur 2D qui a été développé pour répondre à plusieurs besoins. Tout d'abord pour pour pouvoir tester nos algorithmes avant de les lancer sur robot réel ou même GAZEBO afin de s'abstenir des contraintes liés à ces environnements complexes. De plus, il va permettre de pouvoir faire des tests sur l'interprétation géométrique des formules en jeu (courbes de Bézier quadratique, calcul de vitesse/accélération, difficulté des virages,...). Ce simulateur va aussi permettre d'extraire des routes dans un format lu par GAZEBO afin de générer des routes à la volée comformément à notre cahier des charges. Enfin, il a pour objectif d'être utilisé pour faire des simulations d'algorithmes de conduite autonome.

Ce programme a été développé en Python avec la librairie graphique 2D PyGame. L'architecture repose sur un format MVC (Modele - Vue - Controleur) ce qui permet

une indépendance complète des modules graphiques et logiques. Cela permet aussi de destiner cet outil à un futur prometteur grâce à sa facilité de maintenabilité et de compréhension.

Vous trouverez des captures d'écran et une description encore plus détaillé dans le manuel d'utilisation (B) en fin de rapport.

5.1 Architecture globale

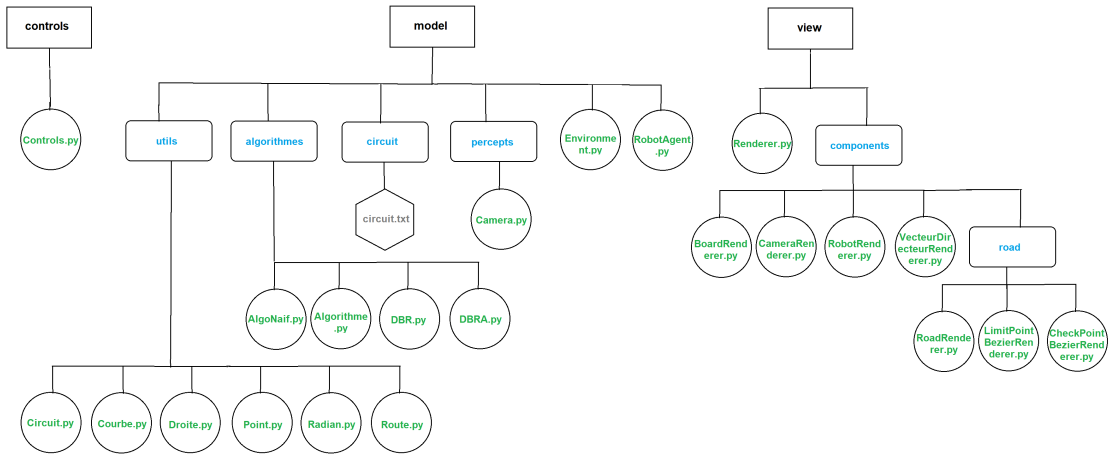


FIGURE 16 – Architecture globale du simulateur

Tout d'abord le point d'entrée du programme est contenu dans la classe `mainLinuxOS.py` et `mainWindowsOS.py` pour les utilisateurs respectifs de ces systèmes. Ces différents point d'entrée ont du être développé en raison de problème de thread lié à PyGame avec Windows. Nous pouvons voir sur la représentation de l'architecture de l'outil ci-contre les trois modules `./model`, `./view` et `./controls`. L'environnement sera représenté dans la classe `Environnement.py` avec une hauteur et largeur défini à 1000 unités. Il est important de noter que l'environnement se lance avec un thread dans `modelThread` avec une vitesse d'évolution de 0.1s. La partie graphique est contenu dans le répertoire `./view` et tous les composants sont contenus dans le répétoire `./view/components/`.Tout comme la partie logique, la vue possède aussi son thread : `RendererThread` qui a un facteur d'actualisation de 0.01s.

Tous les algorithmes d'amélioration de la conduite automatique du robot doivent

être implémenté dans le package `./model/algorithmes` et devront hériter de la classe `Algorithme.py`. Ces algorithmes pourront utiliser les utilitaires géométriques contenu dans le package `./model/utils` comme `Droite.py`,`Point.py`,`Radian.py`,etc... Ils pourront aussi avoir accès aux valeurs du robot comme sa vitesse linéaire/angulaire courante, sa position ou encore sa caméra défini dans `./model/percepts/Camera.py`.

5.2 Calcul de vitesse linéaire et angulaire

5.2.1 Explications mathématiques et algorithmiques

Pour simuler les comportements d'accélération de notre robot dans le simulateur, nous avons utilisés les formules d'accélération linéaire et angulaire suivantes [7].

$$x(t+1) = x_t + v_{x_t} * t + \frac{1}{2} a * t^2$$

$$\theta(t+1) = \theta_t + \omega_{\theta_t} * t + \frac{1}{2} \alpha * t^2$$

Ici, $x(t+1)$ correspond à la position du robot au pas de temps suivant en ayant une vitesse linéaire de v_{x_t} au pas de temps t et une accélération linéaire constante de a. $\theta(t+1)$ correspond à l'angle du robot en radian au pas de temps suivant en utilisant ω_{θ_t} la vitesse angulaire du robot actuelle et α la constante d'accélération angulaire. Cependant, par souci d'implémentation et de test, nous avons gardé une formule en considérant $t = 1$, ce qui nous donne tout de même des résultats satisfaisant.

Par ailleurs, dans le simulateur, nous avons représenté le vecteur directeur du robot par une mesure en radian avant de le transformer en vecteur directeur. Cela nous permet d'effectuer des opérations en radian et de faire une transformation vectorielle de la manière suivante :

$$v = (cos(rad), sin(rad))$$

Ainsi, nous pouvons appliquer la formule d'accélération angulaire cité plus haut pour faire tourner notre robot. Il faut noter aussi que nous faisont des opérations sur les radians modulo 2π rad.

5.2.2 Tests et analyses

Afin de réaliser des tests sur les formules que nous avons cité, nous avons fait tourner un algorithme de rotation en initialisant le vecteur directeur à une valeur de 2π rad et par incrément succéssif de 0.1 rad

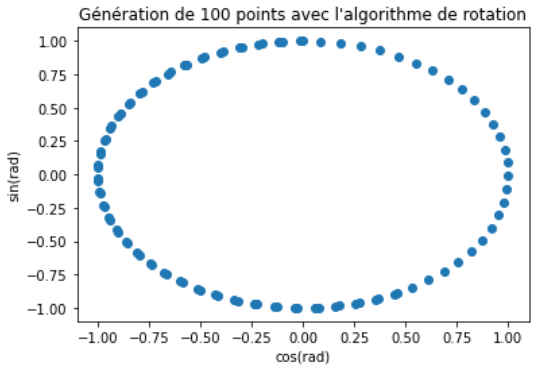


FIGURE 17 – Test de la génération de 100 points avec l'algorithme de rotation

5.3 Calcul de la direction et déplacement du robot

5.3.1 Explications mathématiques et algorithmiques

Pour pouvoir se déplacer et suivre une direction, le robot à besoin d'être capable de se repérer et de s'orienter. Nous allons dans cette partie présenter la procédure de réajustement itérative de la direction du véhicule avec une destination.

FIGURE 18 – Représentation du vecteur directeur du véhicule vers une destination

Comme nous pouvons le voir sur cette figure, nous allons utiliser un vecteur directeur normalisé pour orienter la direction de notre véhicule en fonction de la destination.

Nous allons avoir besoin pour cela de la position du robot notée $R = (x_R, y_R)$, de la position de la destination notée $D = (x_D, y_D)$ et du vecteur directeur de la direction du robot noté $R_I = (x_I, y_I)$ que l'on initialise à $(1, 0)$ comme sur la figure. Nous allons avoir aussi besoins du vecteur directeur V de la droite (R, D) que l'on défini :

$$V = (x_D - x_R, y_D - y_R)$$

On cherche maintenant à minimiser l'écart des vecteurs directeurs entre R_I et V de la manière la plus rapide possible. C'est à dire qu'il faut prendre la décision de soit

tourner vers la gauche ou soit vers la droite le vecteur directeur R_I afin d'atteindre une différence inférieur à un ϵ en radian.

Nous allons utiliser pour cela la formule du calcul d'angle de cosinus :

$$cos(\theta) = \frac{V.R_I}{|V||R_I|}$$

$$\implies \theta = arccos(\frac{V.R_I}{|V||R_I|})$$

Cette formule va alors nous permettre de corriger la direction du vecteur directeur d'un véhicule par itération successive avec l'algorithme suivant :

Algorithm 4 Procédure de réajustement de la direction

Require: r : valeur en radian du vecteur directeur, D : destination, R : position du robot, R_I : vecteur directeur initial, ϵ : marge d'erreur,α : rapidité de rotation (en radian)
$angle_{Init} \leftarrow arccos(\frac{V.R_I}{|V||R_I|})$
$tmp_{rad} \leftarrow (r + \alpha) mod 2 * \pi$
$tmp_{R_I} \leftarrow (cos(tmp_{rad}), sin(tmp_{rad}))$
$angle_{tmp} \leftarrow arccos(\frac{V.tmp_{R_I}}{|V||tmp_{R_I}|})$
if $angle_{Init} * \frac{180}{\pi} < angle_{tmp} * \frac{180}{\pi}$ **then**
  Faire tourner le robot à droite d'un pas de α
else
  Faire tourner le robot à gauche d'un pas de α
end if

Cette procédure va être appelé tout les pas de temps par le robot afin de corriger sa direction. En ce qui concerne la complexité : on enchaîne plusieurs calculs en O(1) sans boucle ni rien de contraignant donc ici la complexité est négligeable.

5.3.2 Tests et analyses

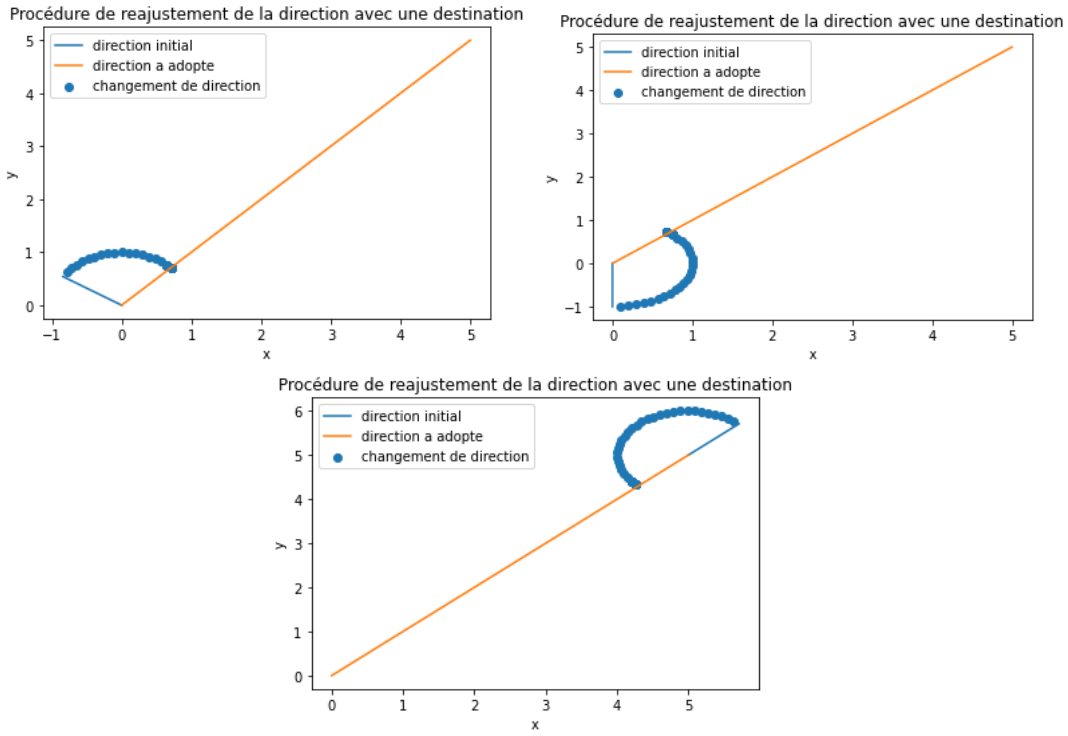


FIGURE 19 – Différentes procédures de réajustement de la direction vers une destination

Pour effectuer des tests sur la procédure de correction de la direction, nous avons pris les valeurs suivantes : $\alpha = 0.1rad$. Nous voyons que sur ces figures, l'algorithme choisit de manière efficace la meilleure direction de rotation quelque soit l'angle entre les deux vecteurs directeurs.

5.4 Conception et géométrie de la route

Afin de dessiner une route, nous disposons de 3 points qui forme une courbe de Bézier. Une fois tracé, la courbe permet de définir le centre de la route. Pour créé les bords de la route, nous récupérons la tangente normalisé en chaque points de la courbe et en déduisons la normale. Cela nous permet de tracer les points à la distance que l'on souhaite selon l'épaisseur de route que l'on veux.

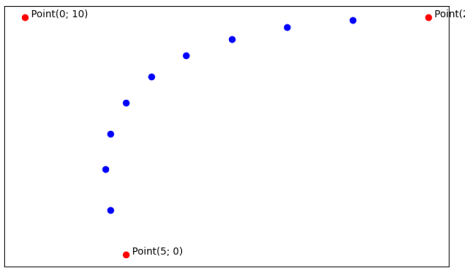


FIGURE 20 – étape 1

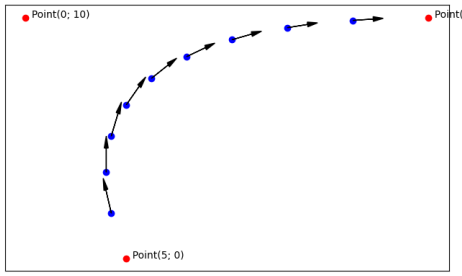


FIGURE 21 – étape 2

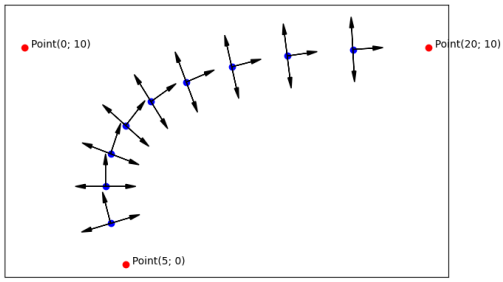


FIGURE 22 – étape 3

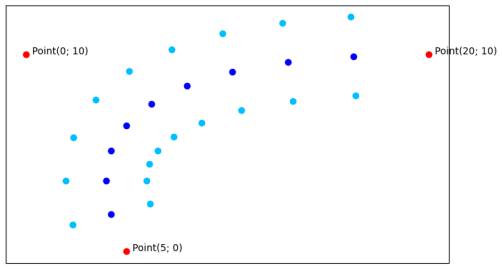


FIGURE 23 – étape 4

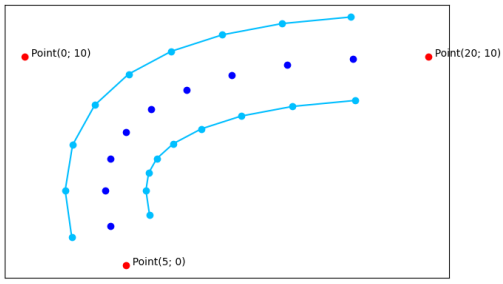


FIGURE 24 – étape 5

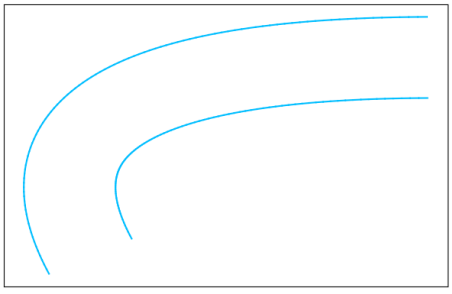


FIGURE 25 – étape 6

FIGURE 26 – Conception de la route par courbe de bézier étape par étape

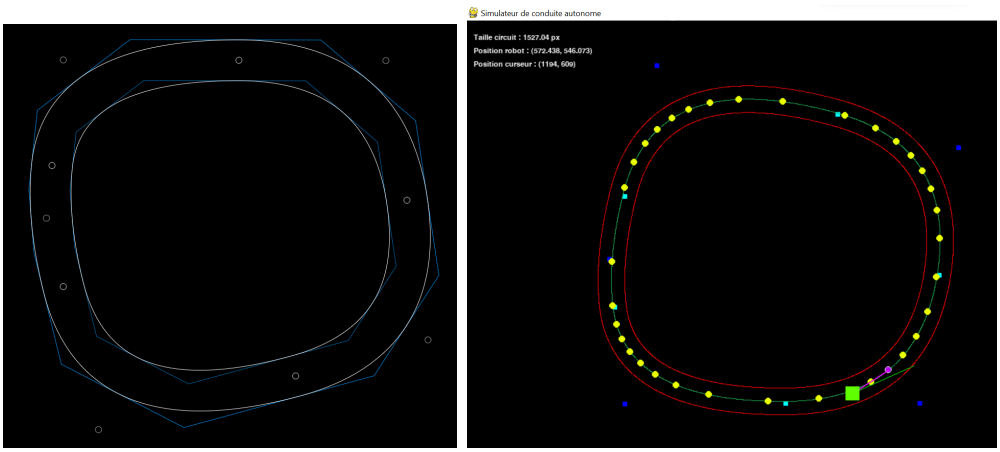
On réalise ce travail sur l'ensemble du circuit, qui n'est qu'une liste de points formant des courbes de Bézier, pour avoir l'ensemble du tracé du circuit.

5.5 Outils et représentation graphique

Dans le simulateur, nous avons représenté l'ensemble des outils nécessaire à la conduite du robot. Tout d'abord, un circuit est dessiné grâce à la technique précédemment énoncé à partir d'un fichier texte où sont écrit l'ensemble des points constituant les courbes de notre circuit. On conserve ensuite l'emplacement des points de contrôles qui ont été créé pour que le robot puisse les utiliser à terme pour la conduite. Le robot est lui dessiné comme un ensemble de trois éléments : La structure du robot sous la forme d'un carré, sa vision (la caméra dans le modèle réel) qui représente son champ de vision de la route ici dessiné comme un rectangle orienté dans son sens de déplacement et enfin une ligne représentant sa direction. Plus de détails sont donnés dans la partie manuel d'utilisation en bas de ce rapport (B) .

5.6 Connections avec Road Map Editor

Pour pouvoir lancer une simulation, notre simulateur à besoin d'avoir des circuits. Cependant, il est difficile de pouvoir générer des coordonnées à la main de manière à représenter un circuit en entier. Pour cela, nous avons utilisé l'outil Road Map Editor qui nous permet de pouvoir créer des circuits avec une interface graphique. Nous avons implémenté un script en Python qui permet de prendre en entrée le fichier XML généré par Road Map Editor et sortir un fichier texte pouvant être lu par notre simulateur.



6 Environnement 3D sur Gazebo :

Nous avons également tenté de porter le travail de génération de routes défini précédemment à un environnement 3D à partir duquel des simulations et expériences pourront être lancées. Les caractéristiques et propriétés des courbes de Bézier ont été soigneusement prises en compte lors du développement de notre générateur qui se base sur un langage à la XML pour définir des routes et ce selon plusieurs modes possibles (Random, prédéfini de manière concise ou encore mix des 2 modes précédents avec certains éléments en random ou présents uniquement selon une probabilité p et d'autres présents à coup sûr). Le langage à la XML est utilisé pour définir des primitives de routes ainsi que leur ordonnancement (modulé par une probabilité de présence de la primitive selon une probabilité p dans le cas du mode Random). Il permet de construire un parcours à partir des "briques" de bases que sont les primitives que nous définirons dans la section suivante.

Pour ce faire, on part d'un fichier écrit en ce langage (respectant les tags spécifiques que nous définirons) et qui définit notre route. Le fichier est parsé en entier pour récupérer les primitives qui y sont définies (dans leur ordre d'apparition) avant de les placer dans un système de coordonnées 2D pour construire une unique route composée des primitives récupérées et soigneusement concaténées pour former une route connectée qui sera exportée dans un fichier XML temporaire avant de passer à la génération du *.sdf* qui sera exploité par gazebo.

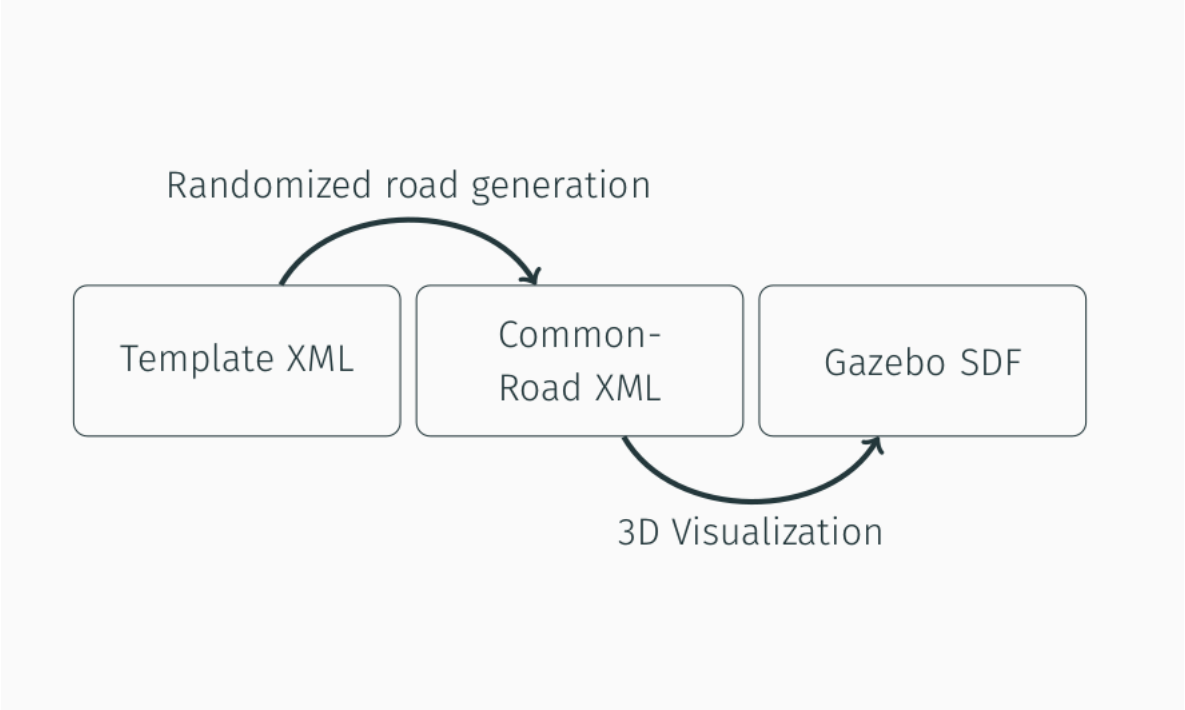


FIGURE 27 – Schema de Génération

6.1 Primitives :

Une primitive consiste en une ou plusieurs bande(s) de route, signe(s) de circulation ou obstacle(s). Chaque primitive est définie dans son propre système de coordonnées dans lequel elle doit avoir un point d'entrée , un point de sortie ainsi que les vecteurs d'entrée et de sortie correspondant à ces points. Le vecteur d'entrée est défini en sens inverse au sens de la route (comme nous le verrons dans les figures qui suivent) et le vecteur de sortie est quant à lui dans le sens de la route. Le point de début et le point de fin sont situés au milieu de la route (partie dashed).

Une description plus détaillée des attributs CSS sera donnée pour chaque primitive.

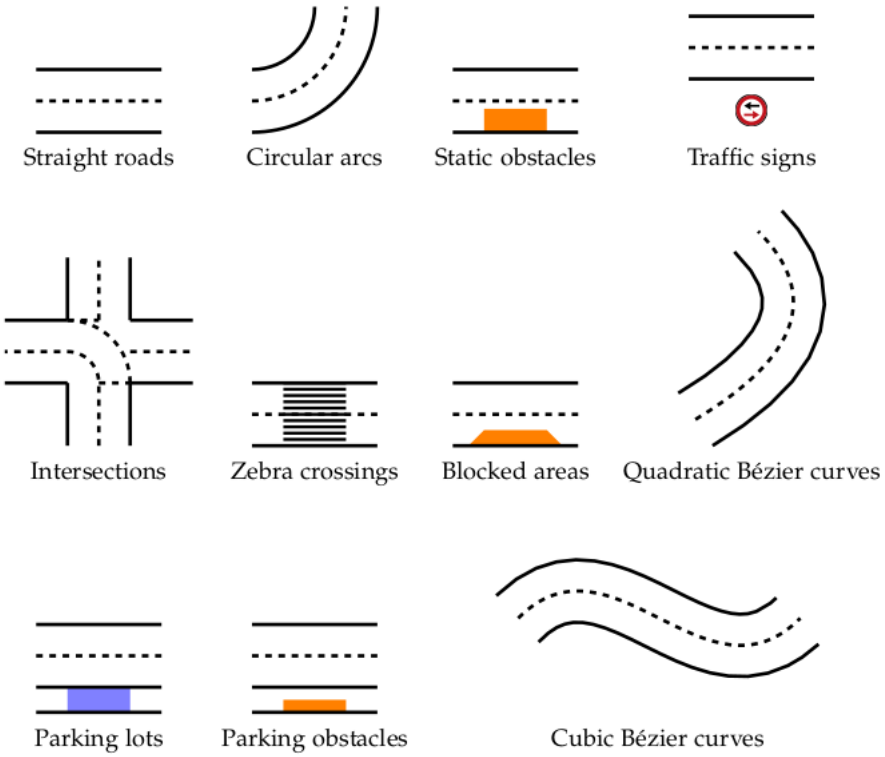


FIGURE 28 – Primitives et objets élémentaires

6.1.1 Straight Roads <line> :

Cette primitive définit une portion de route droite de longueur pouvant être donnée en paramètre (ou définie plus tard lors de la mise à l'échelle). Les bordures de routes peuvent avoir un marking spécifique (dashed, solid ou missing).

<line>			
	length	*required*	float
	leftLine	*optional*	string, default=solid
	middleLine	*optional*	string, default=dashed
	rightLine	*optional*	string, default=solid

Table 3.1.: Definition of the `<line>` element

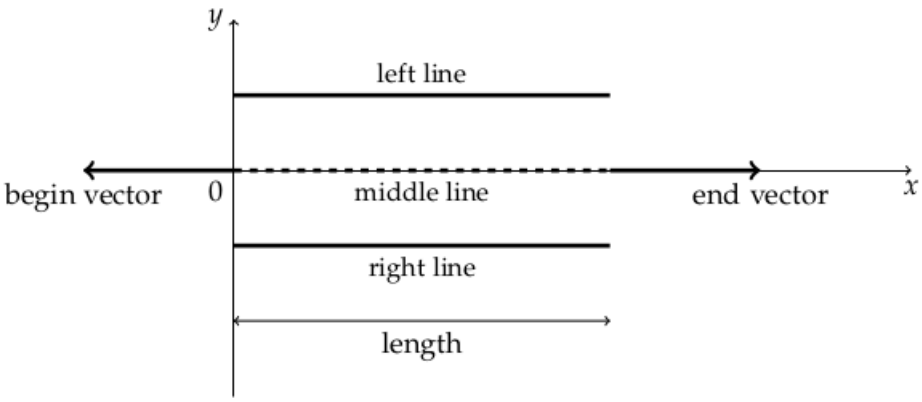


FIGURE 29 – Description <line>

6.1.2 Circular arcs : <leftArc> / <rightArc>

Définition d'une portion courbée selon un angle (en degrés), et un radius donnés en paramètre. On distingue les courbures à droite et les courbures à gauche. Les line markings peuvent être modifiés comme précédemment.

<leftArc>/<rightArc>			
	radius	*required*	float
	angle	*required*	float
	leftLine	*optional*	string, default=solid
	middleLine	*optional*	string, default=dashed
	rightLine	*optional*	string, default=solid

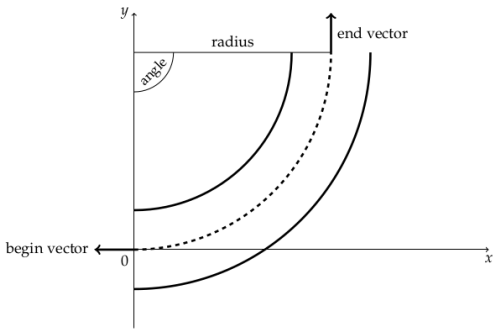


FIGURE 30 – Description arcs circulaires

6.1.3 Static Obstacles :

Un obstacle statique est défini comme une spécialisation de la primitive <line>. La position ainsi que la largeur de l'obstacle peuvent être précisés (anchor : center, position : -0.5 pour l'exemple).

<staticObstacle>			
	width	*required*	float
	length	*required*	float
	position	*required*	float
	anchor	*required*	enum(left, center, right)

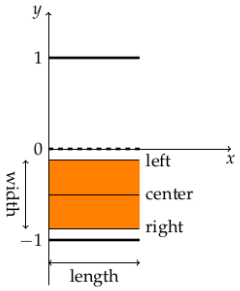


FIGURE 31 – Description <staticObstacle>

6.1.4 Intersections :

L'intersection a été la primitive la plus compliquée à définir, elle comporte deux paramètres *rule* et *turn* qui définissent sa disposition.

rule définit les priorités à respecter au niveau de l'intersection : yield, stop, equal ...

<intersection>			
	rule	*required*	enum(priority-yield, priority-stop, yield, stop, equal)
	turn	*required*	enum(left, straight, right)

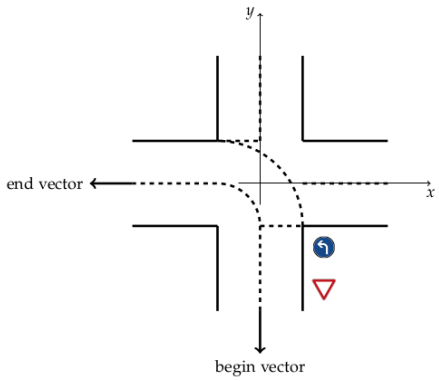
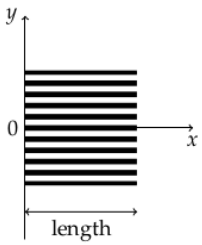


FIGURE 32 – Description <intersection>

6.1.5 Zebra crossings : <zebraCrossing>

Cette primitive est une spécialisation de la primitive <line>. Le passage piéton a pour longueur la longueur totale précisée en argument. On précise que les line markings ne sont pas précisés donc non représentés.



<zebraCrossing>		
length	*required*	float

FIGURE 33 – Description passage piéton

6.1.6 Quadratic Bezier Curve : <quadBezier>

Définition d'une courbe de bézier en fonction de P_0 (start point), P_2 (end point) et d'un point de contrôle P_1. Seuls les points P_1 et P_2 sont définis par l'utilisateur. Le point P_0 est retrouvé automatiquement lors de la génération et les points formant les lignes sont calculés par l'algorithme de De-Casteljau.

<quadBezier>			
	p1x	*required*	float
	p1y	*required*	float
	p2x	*required*	float
	p2y	*required*	float

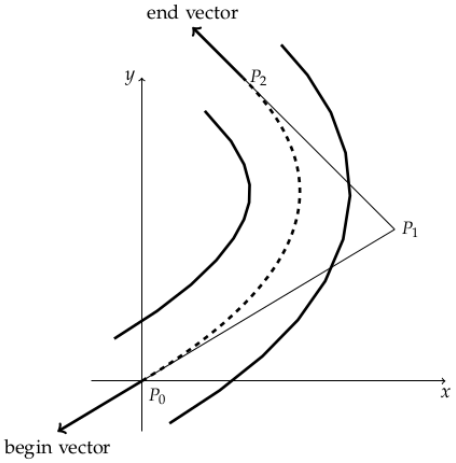


FIGURE 34 – Description <quadBezier>

6.1.7 Cubic Bezier Curve : <cubicBezier>

Les courbes de Bézier cubiques nous permettent de définir des virages en S de manière plus précise que si nous avions à concaténer 2 courbes de Bézier quadratique (<quadBezier>). Malgré de nombreux efforts, des virages consécutifs doivent être définis sous cette forme sous peine de mauvais rendu au moment de la génération du modèle pour Gazebo. Nous avons donc ici 2 points de contrôle et un point final à donner en argument.

<cubicBezier>			
	p1x	*required*	float
	p1y	*required*	float
	p2x	*required*	float
	p2y	*required*	float
	p3x	*required*	float
	p3y	*required*	float

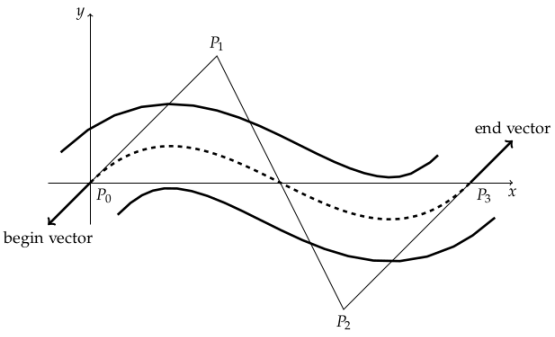


FIGURE 35 – Description <cubicBezier>

6.1.8 Parking lots : <parkingLot>

Spécialisation de la primitive <line> avec cette fois-ci en paramètre la taille de la place de parking.

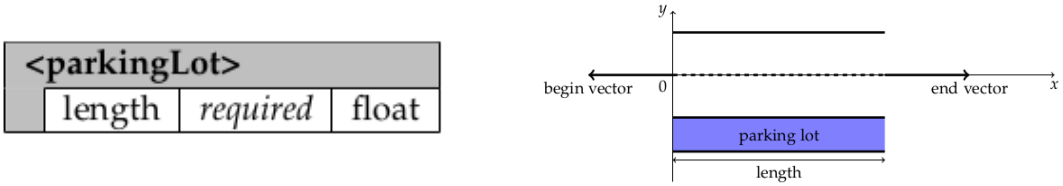


FIGURE 36 – Description <parkingLot>

6.1.9 Traffic Sign : <trafficSign>

Pour cette primitive, les points de début et de fin sont positionnés à $(0, 0)$ et seront recalculés au moment de la génération. Le paneau de signalisation sera positioné à 15 cm de la route la plus proche dans l'ordre de définition du fichier XML défini.

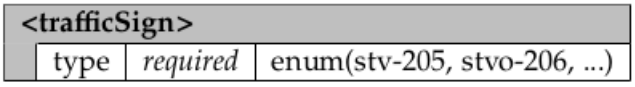


FIGURE 37 – Description panneau de signalisation

6.1.10 Parking Obstacle : <parkingObstacle>

Spécialisation de la primitive <parkingLot> mais un obstacle est positionné à la place de stationnement.

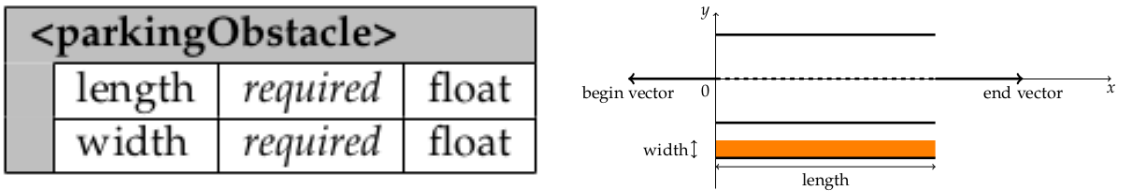


FIGURE 38 – Description obstacle de parking

6.2 Structures de contrôle et assemblage des primitives :

Toutes les primitives définies précédemment sont assemblées dans le XML qui définit notre route. Pour aider à la génération et structurer la route, nous avons défini les structures de contrôle suivantes.

6.2.1 Sequence

Une séquence retourne tous les enfants dans le même ordre tels que définis. Dans cet exemple on a un paneau de signalisation et un passage piéton qui seront récupérés dans cet ordre.

```
<sequence>
    <trafficSign type="stvo-350-10" />
    <zebraCrossing length="0.4" />
</sequence>
```

6.2.2 Optional

Les éléments <optional> retournent aucun, un seul ou plusieurs éléments enfants (si plusieurs sont définis) selon une probabilité p donné en paramètre.

6.2.3 Select + Case

Généralisation de l'élément <optional> à plusieurs enfants. Le poids w correspond à la probabilité d'apparition.

```
<repeat min="1" max="3">
    <select>
        <case w="0.9">
            <trafficSign type="stvo-350-10" />
            <line length="0.4" />
            <zebraCrossing length="0.4" />
        </case>
        <case w="1">
            <trafficSign type="stvo-208" />
            <line length="0.5" />
            <blockedArea width="0.15" length="2" />
        </case>
    </select>
```

```
    <line length="1" />
</repeat>
```

6.2.4 Repeat

Répétion n fois si un nombre n est précisé ou un nombre aléatoire entre un min et un max comme défini dans le bout de XML précédent.

6.2.5 Shuffle

Permutation aléatoire des enfants du tag <shuffle>. Aucun paramètre n'est à préciser pour cette structure de contrôle. Exemple de génération aléatoire de places de parking :

```
<shuffle>
    <sequence>
        <parkingObstacle width="0.28" length="0.2" />
        <parkingLot length="0.55" />
        <parkingObstacle width="0.28" length="0.2" />
    </sequence>
    <sequence>
        <parkingObstacle width="0.28" length="0.2" />
        <parkingLot length="0.63" />
        <parkingObstacle width="0.28" length="0.2" />
    </sequence>
    <sequence>
        <parkingObstacle width="0.28" length="0.2" />
        <parkingLot length="0.70" />
        <parkingObstacle width="0.28" length="0.2" />
    </sequence>
        <parkingObstacle width="0.28" length="0.2" />
        <parkingObstacle width="0.28" length="0.4" />
    <sequence>
        <parkingObstacle width="0.15" length="0.2" />
            <repeat min="3" max="5">
                <optional p="0.8">
                    <select>
                        <case w="1"><parkingLot length="0.10" /></case>
                        <case w="1"><parkingLot length="0.20" /></case>
```

```
                        <case w="1"><parkingLot length="0.40" /></case>
                    </select>
                </optional>
            </repeat>
    </sequence>
</shuffle>
```

6.3 Visualisation :

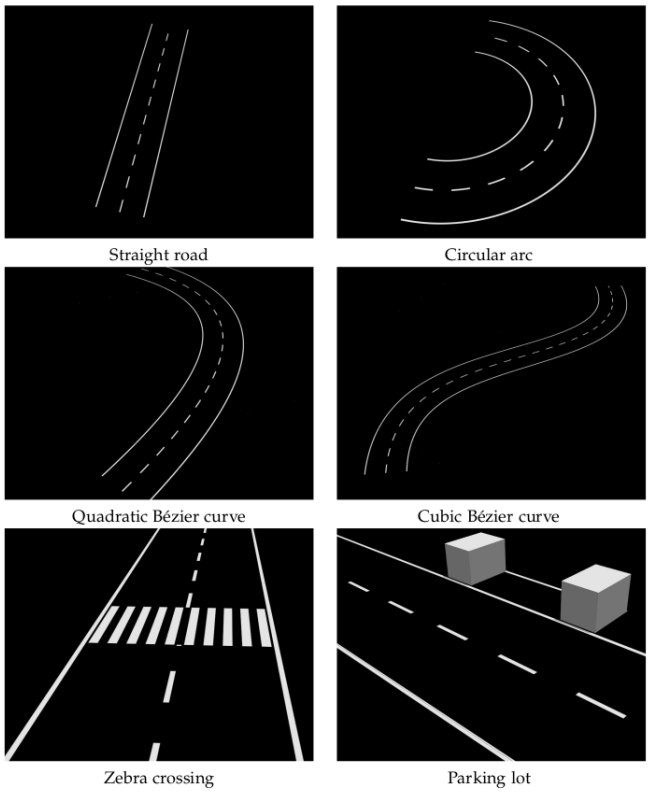


FIGURE 39 – Rendu des différentes primitives

Une fois les éléments du circuit définis dans notre XML de base et le CommonRoad XML temporaire généré, le générateur de rendu gazebo est appelé pour produire le SDF qui sera chargé et lu par Gazebo au lancement. Ce SDF est constitué des éléments de base prédéfinis et communs à toutes les simulations générées. On peut citer l'éclairage avec l'élément <**light**>

```
    <light name='sun_light' type='directional'>
      <pose frame=''>0 0 10 0 -0 0</pose>
      <diffuse>0.5 0.5 0.5 1</diffuse>
      <specular>0.1 0.1 0.1 1</specular>
      <direction>0.1 0.1 -0.9</direction>
      <attenuation>
        <range>20</range>
        <constant>0.5</constant>
        <linear>0.01</linear>
        <quadratic>0.001</quadratic>
      </attenuation>
      <cast_shadows>1</cast_shadows>
    </light>
```

Chaque primitive est dessinée par calcul des points la constituant, génération de la texture associée puis association de cette texture à une **Tile** qui définit la brique de base sur laquelle est contenue la primitive. La texture est dessinée à l'aide du module Pycairo sur lequel nous avons passé un temps conséquent dans la lecture de la documentation et la recherche des fonctions les plus aptes à nous permettre une représentation effective et en accord avec nos modèles de courbes de Bézier. L'association des **Tile** dont la texture a été affectée constitue le circuit résultat (Les **Tile** non modifiées ont un fond noir simple et constituent le **groundplane**, qui est la base sur laquelle roulera le robot.).

Pour des soucis de bonne représentation des interactions physiques, les objets sont tous situés au minimum à 0.1 m de hauteur afin de ne pas être confondus avec le **groundplane**. Conformément à la norme SDF de Gazebo, ils disposent d'une masse (<**mass**/> et d'une matrice d'inertie (<**inertia**/>)

```
<inertial>
    <mass>10</mass>
      <inertia>
        <ixx>0.166667</ixx>
        <ixy>0</ixy>
        <ixz>0</ixz>
        <iyy>0.166667</iyy>
        <iyz>0</iyz>
        <izz>0.166667</izz>
      </inertia>
</inertial>
```

Références

[1] Jean-Michel Ilié, Ahmed-Chawki Chaouche, and François Pêcheux. 2020. E-HoA : A Distributed Layered Architecture for Context-aware Autonomous Vehicles. Procedia Computer Science 170, (2020), 530–538.
DOI :https://doi.org/10.1016/j.procs.2020.03.121

[2] Nell Flaharty, E-HoA Bezier editor : rapport du stage SAR 2020.

[3] 2022. Bézier curve. Wikipedia. Retrieved May 20, 2022 from Courbe de Bézier

[4] Gabriel Suchowolski. 2012. Quadratic bezier through three points and the "equivalent quadratic bezier (theorem)." ResearchGate. Retrieved March 29, 2022 from https://www.researchgate.net/publication/261596712_Quadratic_bezier_through_three_points_and_the_equivalent_quadratic_bezier_theorem.

[5] Donald Ervin Knuth. 1986. The TeXbook. Reading, Mass. : Addison-Wesley. Retrieved May 20, 2022 from http ://archive.org/details/texbook00dona.

[6] Leslie Lamport. 1994. LATEX : A Document Preparation System : User's Guide and Reference Manual. Addison-Wesley.

[7] Chapitre 4.1 – La cinétique de rotation - Amélioration De L'Habitat Et De Réparation. doczz.fr. Retrieved May 20, 2022 from http://doczz.fr/doc/5446596/chapitre-4.1-%E2%80%93-la-cin%C3%A9tique-de-rotation

[8] P. Philipp Bender, J. Ziegler, and C. Stiller. 2014. Lanelets : Efficient map representation for autonomous driving. 2014 IEEE Intelligent Vehicles Symposium Proceedings (2014). DOI :https://doi.org/10.1109/IVS.2014.6856487

[9] Markus Koschi and M. Althoff. 2017. SPOT Manual (Version 2017 a). Retrieved May 20, 2022 from https://www.semanticscholar.org/paper/SPOT-Manual-(-Version-2017-a-)-Koschi-Althoff/fd65c6aa34c0a1c6cfb0d26e6b271c9823a24878

[10] M. Althoff, Markus Koschi, and Stefanie Manzinger. 2017. CommonRoad : Composable benchmarks for motion planning on roads. 2017 IEEE Intelligent Vehicles Symposium (IV) (2017). DOI :https://doi.org/10.1109/IVS.2017.7995802

[11] Gazebo : Tutorial : Gazebo plugins in ROS. Retrieved May 20, 2022 from https://classic.gazebosim.org/tutorials?tut=ros_gzplugins

[12] *The Beauty of Bézier Curves* de Freya Holmér disponible sur Youtube.

[13] *A Primer on Bézier Curves* de Pomax disponible en ligne.

7 Conclusion

Tout d'abord, ce projet nous auras permis d'améliorer nos compétences et nos connaissances en géométrie/mathématique avec l'usage tout au long de ce projet des courbes de Bézier, formule de vitesse à travers l'élaboration d'un algorithme déterministe par méthode de Bézier. Nous avons su monter en compétences en génie logiciel et en développement avec la conception de notre simulateur ce qui à été très enrichissant. Par ailleurs, nous avons appris à gérer le temps d'un projet de cette envergure et nous nous sommes entraidés lorsqu'un de nous à rencontré des difficultés. De plus, nous avons aussi retrouvé dans ce projet, les notions de notre Master ANDROIDE qui se retrouvent à travers des aspects de recherche opérationnelle où l'on souhaite minimiser la durée d'un tour de circuit et le risque de débordement d'une route.

Dans un premier temps, nous sommes fier d'avoir pu répondre à la problématique initiale qui est la conception d'un algorithme déterministe basé sur les courbes de Bézier. Cette approche à su montrer des performances attendu mais aussi des limites non négligeable comme la sortie de route en cas de vitesse excessive. Pour cela, nous recommandons d'utiliser des méthodes par apprentissage pour apprendre les différents paramètres que nous avons du fixer nous mêmes dans nos algorithmes.

Dans un second temps, la transposition du travail en 2D vers un environnement tridimensionnel a été compliquée mais très enrichissante. La modélisation en 2D a été très utile pour fixer de manière précise les points importants et les objectifs à atteindre sans se perdre dans la diversité des problématiques liées à l'ajout d'une dimension. Il a fallu lire énormément de documentation, se renseigner sur un grand nombre de composants et logiciels constituant le framework Gazebo. Le manque de ressources préexistantes nous a contraint à partir de 0 pour créer notre propre environnement et le générateur qui lui est associé afin d'avoir une grande diversité de résultats Malheureusement, le modèle n'est pas tout à fait comme souhaité. il permet cependant de définir de manière malléable une simulation. Il est stable et offre une base riche et conséquente pour une éventuelle ouverture sur un approfondissement de la simulation.

Nous regrettons malheureusement de ne pas avoir su amener le projet jusqu'au but final qui est l'implémentation d'un algorithme déterministe par méthode de Bézier sur robot réel car nous sommes rester dans le cadre de simple simulation.

Plusieurs améliorations sont possible tel que l'utilisation de méthodes par apprentissage comme par exemple discrétiser la route en trois parties (gauche, centre, droit) et corriger son couple (vitesse linéaire, vitesse angulaire) a chaque expérience en fonction de sa position sur les différents segments. Cela nous permet d'apprendre les valeurs critiques de constante d'accélération ou encore utiliser une certaine classification des virages.

8 Remerciements

Nous tenons a remercier et exprimer notre profonde gratitude aux différents professeurs qui nous ont aidé et guidé pour la réalisation de ce projet et notamment Monsieur Jean-Michel Ilié pour le temps qu'il nous a consacré et pour les précieuses informations qu'il nous as prodiguées avec intérêts et compréhension.

ANNEXES

Appendices

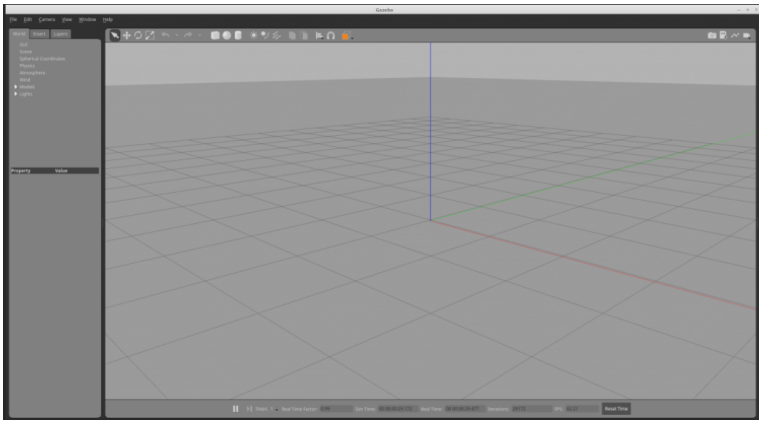
A Cahier des charges

A.1 Description du projet

Ce projet consiste à **améliorer la conduite/les mouvements d'un véhicule autonome dans un espace routier**, en particulier dans sa prise de virages. Nous partirons du principe théorique qu'une trajectoire sur la route pourra être représentée par une suite de courbes de Bézier. Pour ce faire, nous développerons un algorithme déterministe consistant à construire pour chaque virage une courbe de Bézier et à caractériser sa sinuosité suivant l'allure de sa courbe. Le robot du projet E-HoA se déplaçant sur un circuit routier servira de plateforme de test. Il existe aussi un simulateur au nom de GAZEBO, simulateur dans lequel nous allons générer divers routes.

A.2 L'environnement

GAZEBO est un simulateur 3D, cinématique, dynamique et multi-robot permettant de simuler des robots articulés dans des environnements complexes, intérieurs ou extérieurs, réalistes et en trois dimensions. La simulation c'est avant tout un environnement sécurisé et des sensations proches de la conduite automobile, donc l'utilisation de ce simulateur nous permettra d'émuler les conditions réelles et éviter de multiples problèmes rencontrés avec un environnement terrestre tel que la luminosité, le problème de batterie ou sans sacrifier le matériel physique. De plus, valider ses algorithmes dans le simulateur nous permet de confirmer l'efficacité des algorithmes avant de tester dans un contexte réel.



A.3 L'objectif du projet

Le robot a actuellement une perception local de la route et n'arrive pas a prendre de virages. Pour cela, l'objectif principal de ce projet est d'améliorer sa conduite sur sa prise de virage en se focalisant sur les courbes de Béziers.

Dans un premier temps, il s'agira de bien assimiler le concept des courbes de Béziers quadratiques, des différentes méthodes existantes et de prendre connaissances des différentes parties du robot E-HOA ainsi que le simulateur GAZEBO.

Ensuite, nous allons créer un algorithme grâce aux éléments renvoyés par les courbes de Béziers qui permet de pouvoir obtenir un couple (vitesse linéaire, vitesse angulaire) pour pouvoir prendre un virage. De plus, nous allons développer notre propre simulateur en Python qui sera l'intermédiaire entre l'implémentation des algorithmes et la phase de simulation sur GAZEBO. Ce simulateur aura plusieurs objectifs :
- démontrer les calculs géométriques des courbes de bézier et de vitesse,
- analyser les comportements d'un robot en fonction des algorithmes,
- extraire des routes pour la plateforme 3D GAZEBO,
- simuler les comportement de notre robot.

Parallèlement, nous allons aussi générer des routes de manière automatisé sur GAZEBO selon plusieurs modes :
- de manière random,
- de manière prédéfinie,
- ou un mélange des 2 avec des éléments random et des éléments prédéfinis.

A.4 Les outils utilisés

Pour la réalisation de ce projet, nous allons utilisés comme langage de programmation python avec ses bibliothèques très puissantes tel que numpy ou encore pygame. Pour ce qui concerne GAZEBO, le langage principal sera XML ou C++.

A.5 Les contraintes

A.5.1 Matériel a disposition

Pour faire fonctionner le robot réel, il faut que l'environnement soit parfait et un petit problème dû à une luminosité élevée ou encore du Scotch mal collé peut tout

faire planter.

A.5.2 Contraintes liées à l'environnement Gazebo :

Nous n'avons malheureusement pas pu tous bénéficier de l'environnement de test qui est Gazebo. En effet, dû à de nombreux pré-requis comme une certaine version de Linux, des anciennes versions à posséder de ROS et de son écosystème, une personne d'entre nous à été chargé de travailler sur l'élaboration de route sur cet environnement en raison de ses compétences sur cet environnement.

B Manuel utilisateur

TurtleBot3

Les différentes commandes permettent de lancer TurtleBot3 soit de maniére autonome soit a l'aide des touches du clavier.

B.1 Démarches Autorace

Autorace nous permet de faire conduire le TurtleBot de façon autonome, en suivant deux lignes (une jaune et une blanche initialement). Il faut également que tous les éléments soient sur le même réseau (hoa-net).

B.1.1 Raspberry pi 4

Premièrement, nous devons faire un ssh sur la Raspberry pi 4 pour manipuler celle-ci, on peut directement y connecter un clavier et une souris, car elle est équipée d'un écran avec la commande suivante :

`ssh pi@192.168.0.47`
Le mot de passe est le suivant : "raspberry"

On ouvre un premier terminal pour lancer :

```
$ export ROS_MASTER_URI=http://192.168.0.11:11311/
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Optionnellement, pour pouvoir détecter les objets de la route un trapèze a été mis en place. On ouvre un deuxième terminal pour lancer les commandes suivantes :

```
$ source start_openvino.sh
$ rosrun object_detect open_vino_real_time_object_detection.py
```

B.1.2 NVIDIA

La Nvidia Jetson TX2 nous permet de faire de la détection d'objet, détection de ligne avec Autorace et également détecter les obstacles aux alentours avec le LIDAR. Une fois la nvidia branché à l'écran, il faut toujours changer la date, il faut exécuter la commande :

`sudo date mois/jour/heure/minutes/année`

Enfin, on enchaines les différentes commandes :
(premier terminal) : `$ roscore`

(deuxieme terminal) : `$ roslaunch jetson_csi_cam jetson_csi_cam.launch width:= 640 height:= 480 fps:= 10`

(troisieme terminal) : `$ export AUTO_EX_CALIB=action`
`$ roslaunch turtlebot3_autorace_traffic_ligth_camera turtlebot3_autora ce_extrinsic_camera_calibration.launch`

(quatrième terminal) `$ export AUTO_DT_CALIB=calibration`
`$ roslaunch turtlebot3_autorace_traffic_light_detect turtlebot3_autorace_ detect_lane.launch`

(cinquième terminal) `$ roslaunch turtlebot3_autorace_traffic_light_control turtlebot3_autorace_control_lane.launch`

Pour arrêter le robot, il suffit de faire 'CTRL + c'.

B.1.3 Lancement rapide

Pour aller plus rapidement, un fichier launch a été créé.
Pré-requis :
Lancer le bringup sur la Raspberry.
Lancer le trapèze sur la Raspberry.
Lancer :

```
$ roslaunch Commande_robot_detection_ligne.launch
```

B.2 Teleop

parfois, il est préférable de commander le robot avec les touches du clavier, pour cela, après avoir exécuté le roscore sur la nvdia on entre dans un deuxième terminal la commande suivante :

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

B.3 Simulateur de conduite autonome

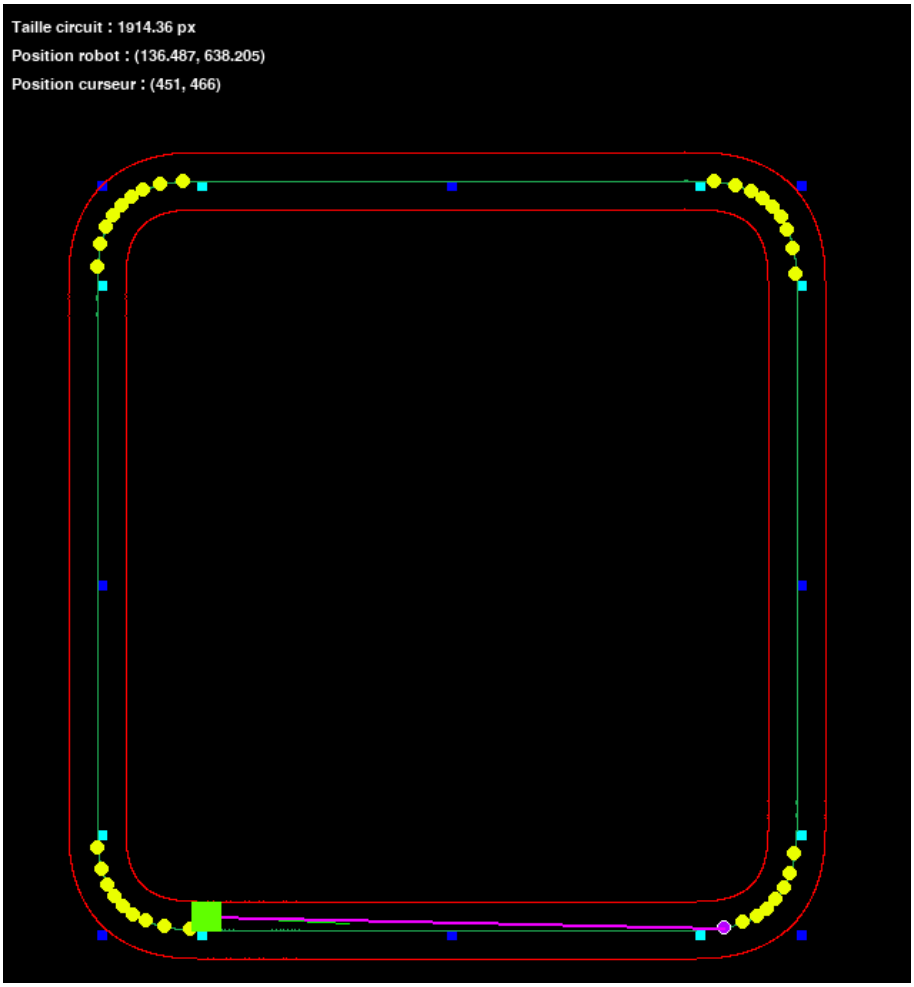


FIGURE 40

B.3.1 Récupération

Afin de configurer votre environnement et pouvoir tester nos implémentations, veuillez suivre les indications suivantes : Avant tout, téléchargez le répertoire contenant nos implémentations de la manière suivante :

```
git clone https://github.com/jdufou1/P-ANDROIDE
```

B.3.2 Lancement du simulateur

Vous aurez besoin de Python3 (Nous avons utilisé la version 3.9.9) pour l'exécution du simulateur ainsi que des librairies **numpy** (version 1.21.5) et **pygame** (version 2.1.2). Déplacez vous ensuite dans le répertoire *phase2*

```
cd phase2
```

puis lancez le simulateur avec une des deux commandes suivante selon votre système d'exploitation.

```
python3 mainLinuxOS.py
```

```
python3 mainWindowsOS.py
```

Le simulateur ce lance avec les paramètres internes au programme. Ces paramètres sont modifiables directement dans le code. Les éléments liés à l'affichage se trouve dans le dossier `./view`. Le fichier `./view/Renderer.py` est l'élément principal de l'affichage initialisant les divers éléments à afficher pendant la simulation tel que le board, le véhicule, ses composants et la route. Vous pourrez ajouter d'autres éléments à afficher, gérer les couleurs des éléments et leur taille. Le dossier `./model` contient les éléments de structures comme le robot, les routes et les outils géométriques. L'environnement regroupe les éléments présents dans la simulation et permettent à ces derniers d'avoir accès aux informations des autres comme par exemple, le robot peu récupérer les points de contrôles une fois qu'ils ont été calculée. On peux également changer les algorithmes de décision du robot dans le dossier algorithmes.

B.3.3 Variables d'environnement

Concernant la partie graphique, nous avons les variables suivantes :

Nom de la variable	Nom du fichier	Utilité
`HEIGHT`	`./view/Renderer.py`	Hauteur de l'interface graphique
`WIDTH`	`./view/Renderer.py`	Largeur de l'interface graphique
`SPEED_VIEW`	`./view/Renderer.py`	Vitesse de rafraîchissement de l'interface graphique

Concernant la partie logique du modèle, nous avons les variables suivantes :

Nom de la variable	Nom du fichier	Utilité
HEIGHT	./model/Environment.py	Hauteur de l'environnement virtuel
WIDTH	./model/Environment.py	Largeur de l'environnement virtuel
filename	./model/Environment.py	Nom du fichier pour l'import de route
SPEED_MODEL	./model/Environment.py	Vitesse de rafraîchissement du modèle

Le robot :

Nom de la variable	Nom du fichier	Utilité
HEIGHT	./model/RobotAgent.py	Hauteur du robot
WIDTH	./model/RobotAgent.py	Largeur du robot
x	./model/RobotAgent.py	Coordonnée x du robot
y	./model/RobotAgent.py	Coordonnée y du robot
algorithme	./model/RobotAgent.py	algorithme contenant une méthode décision exécuté à chaque pas de temps
vlc	./model/RobotAgent.py	Vitesse linéaire courante du robot
vac	./model/RobotAgent.py	Vitesse angulaire courante du robot
current_radian	./model/RobotAgent.py	Angle courant du robot en radian
vecteur_directeur	./model/RobotAgent.py	Vecteur directeur du robot
range_camera	./model/percepts/Camera.py	Portée de la camera du robot
width_camera	./model/percepts/Camera.py	Largeur de couverture de la caméra

Concernant la partie contrôleur, nous avons les variables suivantes :

Nom de la variable	Nom du fichier	Utilité
RESIZABLE	./controls/Controls.py	Extensibilité de la fenêtre
SPEED_CONTROLS	./controls/Controls.py	Vitesse de rafraîchissement des controleurs

B.3.4 Conversion des fichiers XML de Road Map Editor

Road Map Editor nous permet de créer des courbes de manière interactive. Une fois la courbe créée, on l'export au format XML. Ensuite, pour pouvoir transformer ce fichier XML en un fichier txt compatible avec notre simulateur on lance la commande suivante sur le terminal :

```
python3 XML_to_TXT.py fichier.xml
```

Le terminal nous affichera en sortie le nom du nouveau fichier :

```
'circuit_from_xml{cpt}.txt'
```

qu'on pourra retrouver dans le répertoire `./phase2/model/circuit`.
Enfin, pour utiliser ce nouveau circuit il suffit de changer le filename dans le fichier `./phase2/model/Environment.py` par :

```
filename =
os.path.dirname(os.path.abspath(_file_))+"/circuit/circuit_from_xml{cpt}.txt"
```

B.3.5 Ajout et modification du simulateur

Pour implémenter de nouveaux algorithmes de conduite autonomes sur le simulateur, l'utilisateur pourra s'aider de plusieurs méthodes de l'environnement et outils qui ont été implémenté dans cet objectif. Tout d'abord, l'utilisateur devra implémenter les nouveaux algorithmes dans le répertoire `./model/algorithmes/` en créant une classe et en la faisant hériter de la classe `./model/algorithmes/Algorithme.py`. De cette manière ils hériteront de la méthode *decision()* qui sera appelée à chaque pas de temps de la simulation. Un algorithme naïf à été donné comme exemple qui est situé ici : `./model/algorithmes/AlgoNaif.py`

B.3.6 Description des composants graphiques

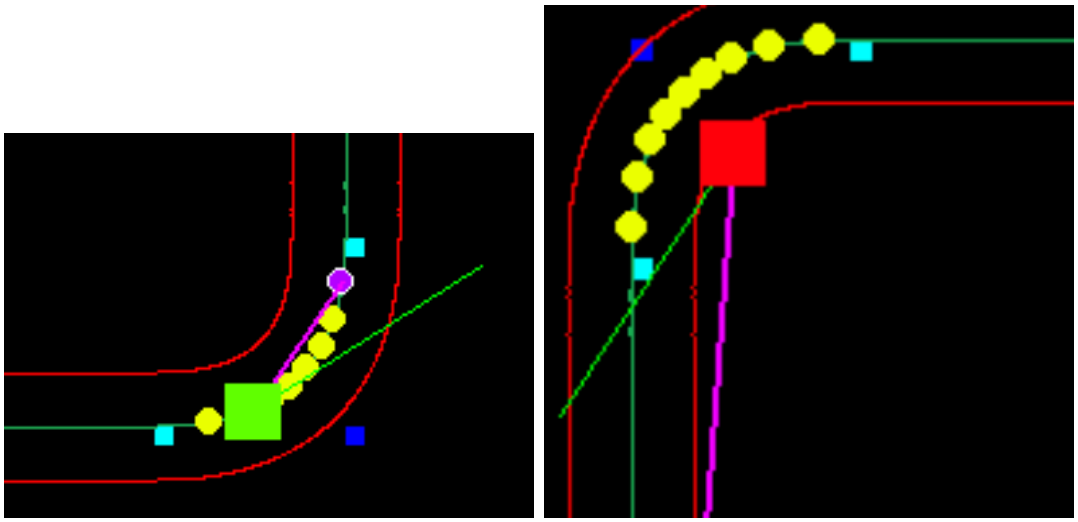


FIGURE 41

Les points bleus :

Nous pouvons voir sur ces figures que les points bleus clairs et bleus foncés sont utilisés pour concevoir un virage de la route par courbe de bézier. Les points bleus clairs représentent les points d'extrémité et le point bleu foncé le point de contrôle.

Les points jaunes :

Les points jaunes représentent l'endroit où l'angle de la courbe de bézier (milieu de la route) est le plus petit. Cela permet aussi au robot de pouvoir se diriger car les coordonnées de ces points sont prient en compte dans l'algorithme que nous avons développé.

Le carré :

Le carré représente le robot. Il est de couleur verte (figure de gauche) quand il est sur le circuit et de couleur rouge (figure de droite) quand il est en dehors.

Les lignes :

La ligne verte qui part du centre du robot correspond au vecteur directeur que celui-ci se sert pour indiquer sa direction. La ligne violette est tracé lorsque le robot suit une direction et colore ainsi le point cible en violet (figure de gauche).

B.4 Gazebo et road generator :

En pré-requis, il faut avoir un environnement python3 avec pour package manager pip afin de pouvoir ajouter les paquets sur lesquels se base notre générateur de routes et de simulations (python3.8 est la version que nous conseillons pour un maximum de compatibilité avec notre code même si nous n'avons eu aucun problème à tester notre générateur sur des versions postérieures).

B.4.1 installation des dépendances :

```
sudo apt-get install python3 gazebo11 libgazebo11-dev
pip install -r requirements.txt
```

Il pourra être nécessaire de downgrader **PyXB** en version 1.2.5 si la version courante (1.2.6) pose un problème au niveau de la génération du XML en B.4.3 .

Une version plus ancienne de Gazebo ne pose pas de problème. La génération des environnements (routes + objets associés) a été testée sur la version 8 de Gazebo et a été réalisée avec succès.

B.4.2 compilation du plugin C++ :

Nous conseillons de bien vérifier que toutes les librairies nécessaires pour ROS soient déjà installée (CMake doit être installé et normalement configuré).

```
cd keyframe_plugin
mkdir build
cd build
cmake ../
make
export GAZEBO_PLUGIN_PATH=`pwd`:$GAZEBO_PLUGIN_PATH
```

B.4.3 Génération du XML au format CommonRoad (étape intermédiaire) :

A partir du fichier **.xml** de votre choix définissant votre route, respectant la norme précisée en 6.0 et présent dans le dossier **presets**/, il vous suffira d'appeler le

script python d'entrée **road-generator.py** qui se chargera de parser votre fichier et de générer un CommonRoad XML.

```
./road-generator.py presets/mon_fichier.xml -o driving-scenario.xml
```

(Il est aussi possible de choisir un autre nom que **driving-scenario.xml** à condition de modifier les commandes qui suivront en conséquence.

B.4.4 Génération du SDF et préparation du rendu Gazebo :

En argument, il faut préciser le nom du dossier où seront générés les fichiers utiles à la simulation (le sdf résultat notamment).

```
./gazebo-renderer.py driving-scenario.xml -o world
```

Si une erreur se produit à ce niveau, il y a de fortes chances que la dernière commande de B.4.2 n'ait pas été prise en compte et que la variable GAZEBO_PLUGIN_PATH ne soit pas clairement définie. Il faut alors la définir en lui associant bien le répertoire **keyframe_plugin**.

B.4.5 Visualisation :

Pour observer l'environnement généré, il suffot d'appeler Gazebo avec le sdf obtenu en paramètre :

```
cd world
gazebo world.sdf
```