TX7676 - Rapport Final

Alexis LEBEL

A24

Table des Matières

Introduction

Le Challenge UTAC est une compétition internationale qui rassemble universités, écoles d'ingénieurs et écoles spécialisées, autour des défis de la mobilité future. Organisée par des acteurs de l'industrie automobile, dont la Société des Ingénieurs de l'Automobile (SIA), cette compétition encourage les étudiants à innover dans des domaines tels que la conduite autonome, la connectivité et la cybersécurité. Les démonstrations et présentations se déroulent sur le circuit de Linas-Montlhéry, offrant une plateforme pour les innovations étudiantes.

Le projet UTonome, porté par l'association éponyme de l'Université de Technologie de Compiègne (UTC), vise à représenter l'UTC lors de ce challenge. L'association, structurée en quatre pôles spécialisés — Simulation, Planification, Ordonnancement et Perception — regroupe des étudiants intéressés par les technologies de mobilité. En tant qu'étudiant, j'ai contribué au développement d'un module de planification et de décision pour un véhicule autonome.

Ce module, intégré dans un système basé sur le framework ROS (Robot Operating System), permet au véhicule de naviguer de manière autonome et sécurisée dans des environnements dynamiques. Il inclut des fonctionnalités telles que la gestion des obstacles, la planification de trajectoires optimales, et la visualisation en temps réel des données de navigation.

Ce rapport présente les étapes de développement du module de planification et de décision, ainsi que les résultats obtenus lors des tests sur simulateur. Il détaille les choix techniques, les algorithmes implémentés, et les perspectives d'amélioration pour les prochaines itérations du projet.

1 Etat de l'Art

1.1 Travaux Précédents

Le projet UTonome 2024 (P24) a été réalisé en utilisant MATLAB comme environnement de développement principal. Ce choix a permis d'exploiter les capacités de simulation et de traitement de données offertes par ce logiciel.

1.1.1 Simulation de véhicules avec le simulateur CARLA

Pour tester et valider les algorithmes développés, le simulateur CARLA a été utilisé. CARLA est un simulateur de conduite autonome open-source qui permet de créer des scénarios réalistes et de tester les performances des véhicules autonomes dans des environnements virtuels.

1.1.2 Module de planification

Le module de planification est basé sur l'algorithme A*. Cet algorithme est utilisé pour rechercher le plus court chemin entre un point de départ et une destination, en tenant compte des contraintes de l'environnement. L'algorithme A* a été choisi pour son efficacité et sa capacité à trouver des chemins optimaux dans des environnements complexes.

1.1.3 Module de décision/contrôle

Le module de décision/contrôle repose sur le concept de "contrôle par waypoints". Ce concept permet de définir des points de passage que le véhicule doit suivre pour atteindre sa destination. Trois modes de conduite ont été implémentés :

1. **Suivi de cible (loi d'attraction)** : Ce mode permet au véhicule de suivre une cible mobile en utilisant une loi d'attraction. La cible peut être un autre véhicule ou un point de passage défini par le module de planification.
2. **Adaptive Cruise Control (ACC)** : Ce mode permet au véhicule de maintenir une distance de sécurité avec le véhicule qui le précède. L'ACC ajuste automatiquement la vitesse du véhicule en fonction de la distance et de la vitesse relative du véhicule précédent.
3. **Évitement d'obstacles** : Ce mode permet au véhicule de détecter et d'éviter les obstacles sur sa trajectoire. Les obstacles peuvent être des véhicules, des piétons ou des objets statiques.

1.1.4 Module de perception

Le module de perception est responsable de la détection des panneaux de signalisation en temps réel. Un algorithme léger a été développé pour permettre une détection rapide et précise des panneaux. Ce module est essentiel pour assurer la sécurité et le respect des règles de circulation par le véhicule autonome.

1.2 Intégration sur un véhicule réel

1.2.1 Conversion en Simulink

Dans le cadre de l'intégration du module de planification/décision sur la Renault Zoé, le projet a été entièrement converti en Simulink pour tirer parti des capacités de simulation et de modélisation. Cette conversion visait à faciliter le développement et le test des algorithmes de planification et de décision dans un environnement contrôlé avant leur déploiement sur le véhicule. L'objectif final étant de créer un module que l'on peut compiler et exécuter directement sur le véhicule réel (Matlab et Simulink Coder).

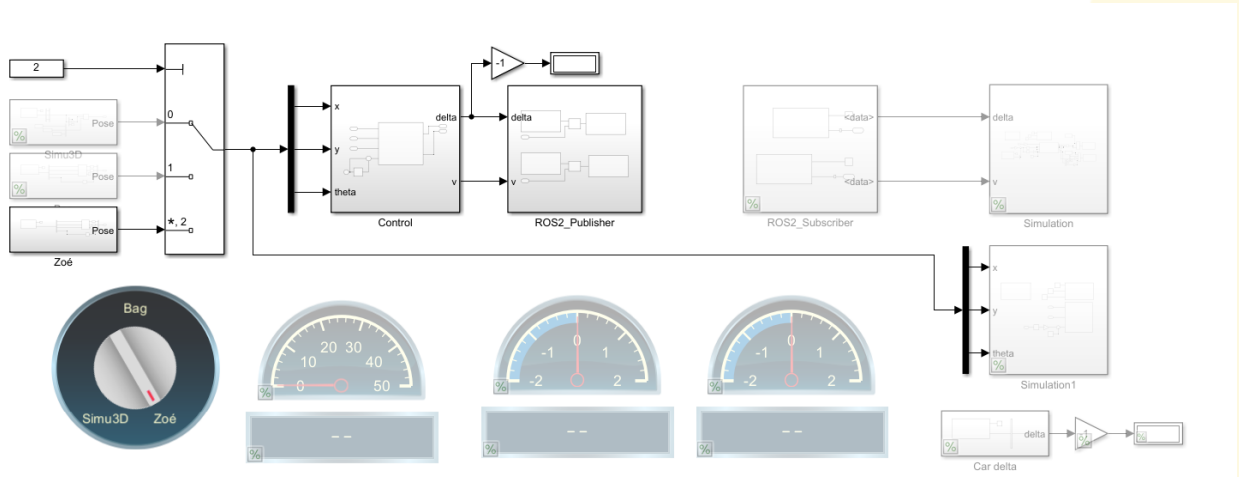

Figure 1: Aperçu du modèle Simulink en mode "Contrôle"

Des blocs Simulink spécifiques ont été utilisés pour gérer les interactions avec le Robot Operating System (ROS), permettant ainsi de publier et de souscrire à des topics ROS. Cette intégration était essentielle pour assurer une communication fluide entre le module de planification/décision et le système de contrôle du véhicule.

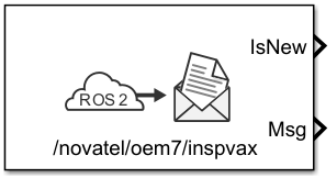

Figure 2: Block d'abonnement ROS2

Le modèle Simulink a été créé en utilisant la classe **AutonomousVehicle**, partie de la dernière édition du projet UTonome. Cette classe a été modifiée pour être conforme aux spécifications concernant l'exécution en temps réel, et la génération et compilation de code C++ pour le déploiement sur le véhicule.

Autour, le modèle Simulink a été conçu pour être modulaire, pouvant être configuré en mode "Simulation" ou "Contrôle de véhicule réel". Cette modularité permet de tester les algorithmes de planification et de décision

dans un environnement simulé, puis de les déployer sur le véhicule réel pour des essais sur route. **Cette transformation a été réalisée en collaboration avec Cyril PETER, président de l'association UTonome.**

1.2.2 Challenge rencontré

Cependant, des problèmes de connexion sont survenus lors des essais d'intégration entre le projet UTonome et le véhicule. Pour identifier la source du problème, plusieurs configurations ont été testées. Une première configuration consistait à installer MATLAB sur un ordinateur à part, connecté au véhicule. Cette approche visait à isoler les environnements de développement et d'exécution, mais des problèmes de latence et de synchronisation ont été observés, rendant la communication inefficace. Parfois, la connexion entre les deux systèmes était complétement inopérante (Erreur `Transport Stopped`).

Une seconde configuration, avec MATLAB installé directement sur l'ordinateur de bord du véhicule, a permis de réduire les problèmes de latence. Cependant, cette configuration a révélé une incompatibilité entre les versions de ROS utilisées par MATLAB (Humble) et celles installées sur le véhicule (Jazzy). Cette incompatibilité a empêché une communication fluide entre les deux systèmes, rendant nécessaire une solution alternative.

Pour surmonter ces incompatibilités, il a été décidé de développer la partie génération de waypoints et contrôle en Python. Cette approche permet de bénéficier de la flexibilité et de la compatibilité accrue de Python avec les différentes versions de ROS. Les avantages de Matlab ne sont pas perdus pour autant, puisqu'il est possible d'exécuter du code Python depuis Matlab, et de passer des données entre les deux environnements.

Cette solution a permis de résoudre les problèmes de compatibilité tout en assurant une intégration efficace du module de planification/décision sur la Renault Zoé. Les essais ultérieurs ont montré une amélioration significative de la communication entre le module et le système de contrôle du véhicule, permettant ainsi de valider l'approche adoptée.

2 Adaptation du module de planification

2.1 Architecture du module

Le module de planification/décision est conçu pour permettre une navigation sûre et efficace dans des environnements dynamiques. L'architecture est composée de plusieurs sous-systèmes interconnectés, chacun ayant un rôle spécifique dans la réalisation de cet objectif.

Planification de Trajectoire :

La planification de trajectoire est initialement réalisée par un nœud externe qui fournit les waypoints de la carte. Ces waypoints définissent le chemin global que le véhicule doit suivre pour atteindre sa destination. Le rôle du module est de transformer ces waypoints en une trajectoire exécutable par le véhicule.

Évitement d'Obstacles :

L'évitement d'obstacles est réalisé par les outils fournis dans `AvoidanceToolset`. Ces outils détectent et contournent les obstacles en utilisant des ensembles de coordonnées pour représenter les positions des obstacles et du véhicule. Les algorithmes d'évitement d'obstacles calculent des trajectoires alternatives en temps réel, permettant au véhicule de naviguer en toute sécurité dans des environnements encombrés. La détection d'obstacles est réalisée en intégrant des données de capteurs, telles que les lidars ou les caméras, pour identifier les obstacles potentiels sur le chemin du véhicule.

Traitement de Chemin :

Le traitement des points de chemin est effectué par `EnhancedPathProcessor`, qui utilise des splines cubiques pour interpoler les waypoints fournis par le nœud externe. Cette interpolation permet de générer des trajectoires lisses et continues, améliorant ainsi la stabilité et le confort de conduite. Le processeur de chemin peut également ajuster dynamiquement les trajectoires en fonction des conditions environnementales ou des obstacles détectés, en effectuant une rediscrétisation des waypoints pour une navigation plus fluide.

Intégration et Communication :

Le nœud ROS agit comme le centre de communication du système. Il reçoit les waypoints de la carte, les positions des obstacles, et transmet les commandes de contrôle au véhicule. L'utilisation de ROS permet une intégration facile avec d'autres modules et capteurs, facilitant l'extensibilité du système. Le nœud ROS gère également la synchronisation des données entre les différents composants, assurant une coordination fluide et en temps réel.

Visualisation et Débogage :

Les outils de visualisation fournissent des moyens de tracer les trajectoires et les obstacles, facilitant ainsi le débogage et l'analyse des performances du système. Les tracés permettent aux développeurs de visualiser les décisions prises par le système de planification et d'évitement d'obstacles, ce qui est crucial pour l'amélioration continue et la validation des algorithmes.

Intégration Globale :

L'architecture globale du module est conçue pour être modulaire et extensible. Chaque composant joue un rôle spécifique dans la planification et l'exécution des trajectoires, tout en étant capable de fonctionner

de manière autonome. Cette modularité permet d'ajouter ou de modifier des fonctionnalités sans affecter l'ensemble du système.

Les données des capteurs sont traitées par `AvoidanceToolset` pour détecter les obstacles. Le nœud ROS utilise ces informations pour générer des trajectoires sûres à partir des waypoints fournis. `Enhanced-PathProcessor` affine ces trajectoires pour assurer une navigation fluide. Le nœud ROS coordonne ces interactions et transmet les commandes finales au véhicule.

Cette architecture robuste et flexible permet au véhicule autonome de naviguer efficacement dans divers environnements, tout en assurant la sécurité et le confort des passagers.

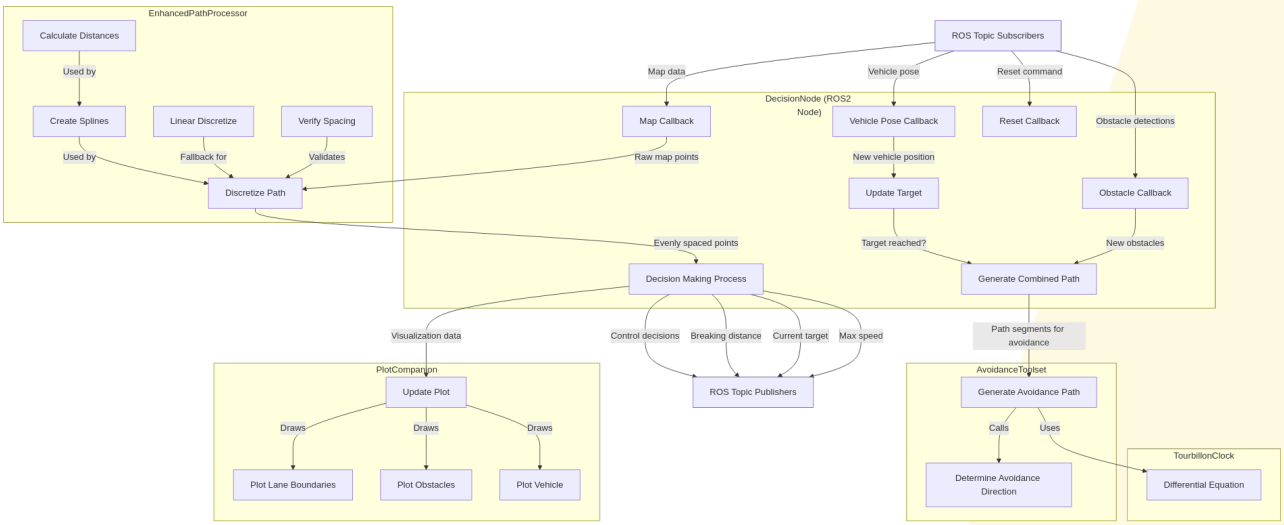

Figure 3: Architecture Globale du module de Décision

2.2 Planification de trajectoires

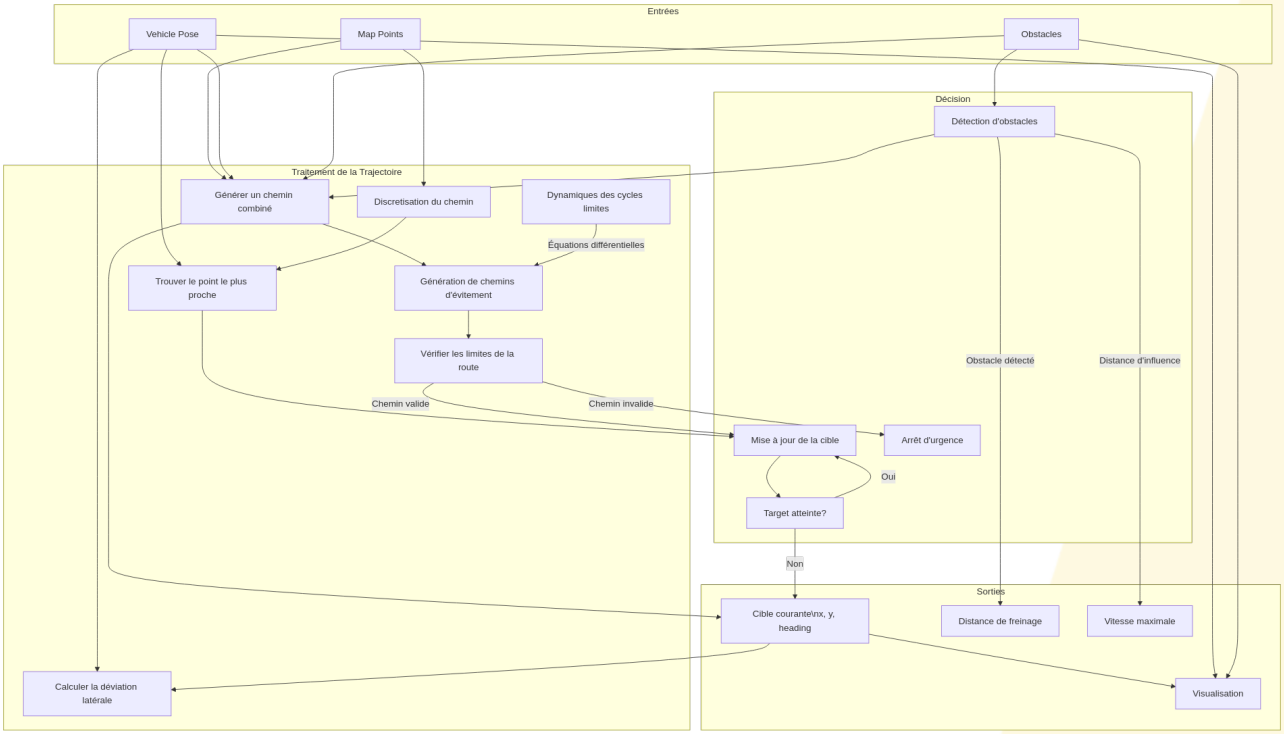

Figure 4: Flowchart du module de trajectoire

La planification de trajectoires est une composante essentielle du module de planification et de décision pour le véhicule autonome. Cette section détaille les méthodes et algorithmes utilisés pour générer des trajectoires sûres et optimales, permettant au véhicule de naviguer efficacement dans des environnements dynamiques.

2.2.1 Génération de Waypoints

La génération de waypoints est la première étape de la planification de trajectoires. Les waypoints sont des points de passage que le véhicule doit suivre pour atteindre sa destination. Ces points sont déterminés par un nœud externe qui fournit une carte globale du trajet. L'algorithme de planification utilise ces waypoints pour créer une trajectoire initiale.

Dans notre implémentation, les waypoints proviennent initialement de la carte de Seville, transmise via le topic ROS 'map_seville'. La fonction `map_seville_callback` traite ces données brutes et applique un processus de discrétisation pour garantir un espacement uniforme entre les points :

```
def map_seville_callback(self, map_seville):
    if not self.seville_enu_points:
        # Extract original points
        original_points = [(point.x, point.y) for point in map_seville.points]

        # Create path processor instance
        path_processor = EnhancedPathProcessor()
```

```
        # Discretize the path with spacing
        discretized_path = path_processor.discretize_path(
            points=original_points,
            spacing=self.get_parameter('path_discretization_distance').value,
            periodic=True  # Assuming the path is a closed loop
        )
```

Cette discrétisation uniforme permet d'établir des estimations cohérentes de distance, de vitesse et d'accélération.

2.2.2 Interpolation par Splines Cubiques

Pour assurer une navigation fluide et stable, les waypoints sont interpolés à l'aide de splines cubiques. L'outil `EnhancedPathProcessor` est utilisé pour cette tâche. Les splines cubiques permettent de générer des trajectoires lisses et continues, évitant ainsi les mouvements brusques et améliorant le confort de conduite. L'interpolation prend en compte les contraintes dynamiques de l'environnement, telles que la présence d'obstacles ou les conditions de la route.

Les splines cubiques sont générées en utilisant des polynômes de degré trois pour interpoler les waypoints fournis. Chaque segment de la spline entre deux waypoints ((x_i, y_i)) et ((x_{i+1}, y_{i+1})) est défini par une équation cubique de la forme :

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

Dans l'implémentation, les splines sont créées à l'aide de la fonction `_create_splines` de la classe `EnhancedPathProcessor` :

```
def _create_splines(self, points: List[Tuple[float, float]], periodic: bool = True):
    x_coords, y_coords = zip(*points)
    x_coords = np.array(x_coords)
    y_coords = np.array(y_coords)

    # Use cumulative distance for parameterization
    t = self._calculate_cumulative_distances(points)

    if periodic:
        # Add periodic point for closed loops
        t = np.append(t, t[-1] + math.sqrt(
            (x_coords[0] - x_coords[-1])**2 +
            (y_coords[0] - y_coords[-1])**2
        ))
        x_coords = np.append(x_coords, x_coords[0])
        y_coords = np.append(y_coords, y_coords[0])

    # Create splines using cumulative distance parameterization
    self.spline_x = CubicSpline(t, x_coords, bc_type='periodic' if periodic else 'natural')
    self.spline_y = CubicSpline(t, y_coords, bc_type='periodic' if periodic else 'natural')
    self.path_length = float(t[-1])
```

2.2.3 Ajustement Dynamique des Trajectoires

Les trajectoires générées sont continuellement ajustées en fonction des conditions environnementales et des obstacles détectés. L'outil `AvoidanceToolset` intègre les données d'obstacles retournés par les modules de perception pour calculer des trajectoires alternatives en temps réel, en utilisant la méthode des cycles limites. Cette approche permet au véhicule de naviguer en toute sécurité dans des environnements encombrés.

La méthode des cycles limites, implémentée dans la classe `TourbillonClock`, est basée sur des systèmes dynamiques non linéaires qui génèrent des trajectoires stables autour d'un point d'attraction. L'équation différentielle régissant ce comportement est donnée par :

$$\dot{x} = y + x \cdot \mu(r^2 - x^2 - y^2)$$

$$\dot{y} = -x + y \cdot \mu(r^2 - x^2 - y^2)$$

où (x, y) est la position actuelle, r est le rayon du cycle limite, et μ est un paramètre contrôlant la convergence. Ces équations sont implémentées dans la méthode `equation_diff_tourbillon_clock` :

```
def equation_diff_tourbillon_clock(self, vector: List[float], radius_limit_cycle: float) ->
↪  np.ndarray:
    x, y = vector

    # Calculate common term for both derivatives
    common_term = self.mu * (radius_limit_cycle**2 - x**2 - y**2)

    # Calculate derivatives
    x_dot = y + x * common_term
    y_dot = -x + y * common_term

    return np.array([x_dot, y_dot])
```

Si la trajectoire candidate avec évitement sort de la voie, elle est considérée comme invalide. Dans ce cas, le système passe en état "Arrêt d'urgence" et arrête le véhicule en publiant une distance de freinage à 0 (que le contrôleur de vitesse interprète comme un freinage d'urgence).

Le module vérifie que la trajectoire générée respecte les limites de la voie à l'aide de la méthode `check_path_within_boundaries` :

```
def check_path_within_boundaries(self,
                           avoidance_path: List[CoordinatesSet],
                           seville_points: List[CoordinatesSet],
                           lane_left_gap: float,
                           lane_right_gap: float) -> Tuple[bool, List[int]]:
    """
    Check if all points in the avoidance path stay within road boundaries.
    """
    # ...
    # Calculate lateral deviation
    lateral_dev = (target_vector['x'] * path_unit['y'] -
                target_vector['y'] * path_unit['x'])

    # Check if point is outside boundaries
    if lateral_dev < -lane_left_gap or lateral_dev > lane_right_gap:
```

```
            violation_indices.append(i)
        # ...
```

Si la trajectoire d'évitement proposée sort des limites autorisées, le système active l'arrêt d'urgence :

```
if not in_bounds:
    self.get_logger().warning('Avoidance path violates road boundaries - skipping')
    self.obstacle_to_break = obstacle_pos
    self.obstacle_stop = True
    continue
```

2.2.4 Fourniture de la cible au contrôleur

Une fois la trajectoire générée, une étape de map matching est réalisée pour déterminer le waypoint le plus proche du véhicule. Cette opération est accomplie via la fonction `get_nearest_point` qui utilise une combinaison de distance euclidienne et d'orientation relative pour identifier le point le plus pertinent :

```
def get_nearest_point(self, vehiclePose, sevilleMap) -> int:
    minDistance = 1000000000
    nearestPointId = 0
    for i in range(len(sevilleMap)):
        error_heading = abs(self.soustraction_angles_atan2(vehiclePose.heading,
↪   sevilleMap[i].heading))
        if (error_heading<0.5):
            distance = self.distance(sevilleMap[i], vehiclePose)
            if distance < minDistance:
                minDistance = distance
                nearestPointId = i
    return nearestPointId
```

La particularité de cette approche est la condition sur l'erreur d'orientation (`error_heading < 0.5`), qui garantit que le point choisi correspond non seulement à la position mais aussi à la direction de déplacement du véhicule. Cette contrainte additionnelle est essentielle pour éviter les comportements oscillatoires ou les sauts de points lors de croisements ou de boucles dans la trajectoire.

Ensuite, la cible, définie comme le point suivant de la trajectoire, est fournie au contrôleur pour le suivi. Le système publie cette cible sur le topic ROS 'current_target' :

```
# Publish current target
target_msg = Point()
next_target = self.current_path[(self.path_index + 4) % len(self.current_path)]
target_msg.x = next_target.x
target_msg.y = next_target.y
target_msg.z = next_target.heading
self.current_target_pub.publish(target_msg)
```

Notez l'utilisation de `(self.path_index + 4) % len(self.current_path)` pour choisir un point légèrement en avance sur la position actuelle. Ce "look-ahead" permet d'anticiper les virages et d'assurer une conduite plus fluide et naturelle.

2.3 Gestion des obstacles

La gestion des obstacles est un processus essentiel pour garantir la sécurité et l'efficacité du déplacement autonome. Ce processus peut être décomposé en plusieurs étapes clés :

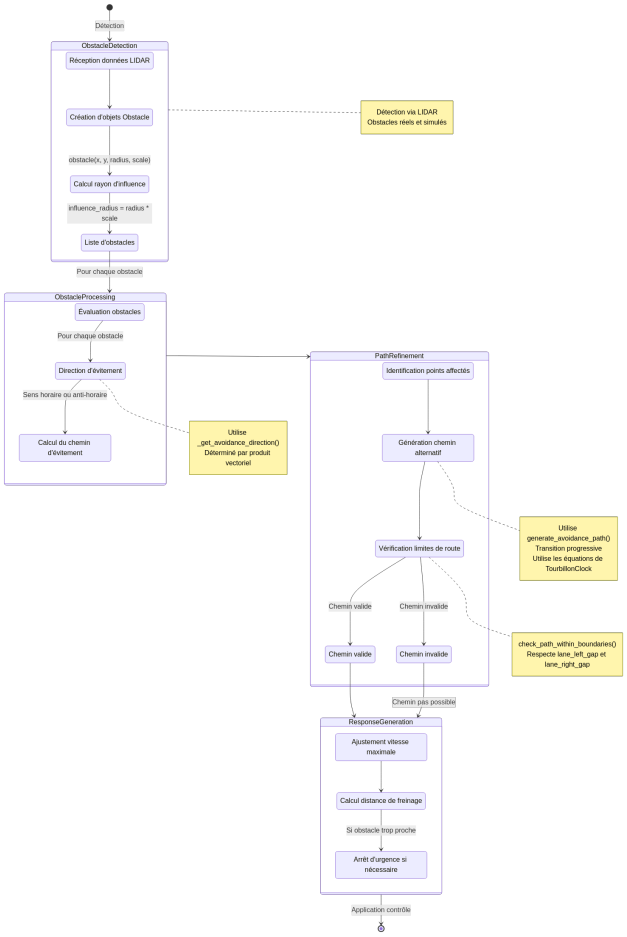

Figure 5: Flowchart de la gestion des obstacles

2.3.1 Détection des Obstacles

La détection des obstacles commence par la réception des données du module de perception via le callback `obstacle_callback` :

```
def obstacle_callback(self, msg):
    """
    Callback for receiving and processing obstacle data.
    """
    self.get_logger().info("Obstacle data received!")
    # Extract obstacle data
    self.obstacles = []

    # message is point array
    for pedestrian in msg.distances:
        self.obstacles.append(Obstacle(pedestrian.center.x, pedestrian.center.y, 1, 1.8))
    if self.fake_obstacle:
        self.obstacles.append(self.fake_obstacle)
```

Chaque obstacle est modélisé comme un objet `Obstacle` avec quatre attributs principaux : - Position (x, y) : Coordonnées du centre de l'obstacle - Rayon physique : Dimension réelle de l'obstacle - Facteur d'échelle : Multiplicateur appliqué au rayon pour définir une zone d'influence

Le rayon d'influence, calculé comme `radius * scale`, définit une zone étendue autour de l'obstacle que

le véhicule devrait éviter. Cette approche permet d'ajuster la marge de sécurité en fonction du type d'obstacle et du niveau de risque associé.

2.3.2 Traitement des Obstacles

Une fois les obstacles détectés, chaque obstacle est évalué pour déterminer son impact potentiel sur le chemin actuel. La direction d'évitement est calculée en utilisant la fonction `_get_avoidance_direction()` dans la classe `AvoidanceToolset` :

```
def _get_avoidance_direction(self, obstacle: Obstacle) -> int:
    """
    Determine the preferred avoidance direction (clockwise or counterclockwise)
    Returns 1 for clockwise, -1 for counterclockwise
    """
    # Vector from obstacle to target
    to_target = np.array([
        self.currentTargetPose.x - obstacle.x,
        self.currentTargetPose.y - obstacle.y
    ])

    # Vector from obstacle to vehicle
    to_vehicle = np.array([
        self.vehiclePose.x - obstacle.x,
        self.vehiclePose.y - obstacle.y
    ])

    # Cross product to determine which side the vehicle is on
    cross_product = np.cross(to_target, to_vehicle)
    return 1 if cross_product < 0 else -1
```

Cette méthode utilise le produit vectoriel pour déterminer si l'évitement doit se faire dans le sens horaire ou anti-horaire. La direction choisie est celle qui minimise la déviation par rapport à la trajectoire désirée vers la cible. Mathématiquement, le produit vectoriel $\vec{v} \times \vec{w}$ est positif si $\vec{w}$ est à gauche de $\vec{v}$, et négatif s'il est à droite. Cette propriété géométrique est exploitée pour choisir la direction d'évitement optimale.

Ensuite, un chemin d'évitement est calculé pour contourner l'obstacle de manière progressive, en utilisant la méthode `generate_avoidance_path` :

```
def generate_avoidance_path(self,
                            obstacle: Obstacle,
                            d_dist: float = 0.1,
                            max_steps: int = 1000) -> List[Target]:
    # ...
    for _ in range(max_steps):
        # Calculate influence factor with earlier, smoother transition
        dist_ratio = (dist_to_obstacle / desired_distance) * 2

        # Smooth transition based on distance
        transition_factor = np.clip((transition_start - dist_to_obstacle) / (transition_start -
↪  desired_distance), 0, 1)
        transition_factor = np.sin(transition_factor * np.pi / 2)  # Smooth sigmoid-like curve

        influence = transition_factor * np.exp(-2.0 * (dist_ratio - 1.0) ** 2)
        influence = np.clip(influence, 0, 1) * 1.3

        # Direct path component
        direct_vel = to_target / (dist_to_target + 1e-6)
```

```
        # Avoidance component
        if dist_to_obstacle > 0:
            # Normalized vector from obstacle to current position
            radial = to_obstacle / (dist_to_obstacle + 1e-6)

            # Tangential direction for avoidance
            tangent = np.array([-radial[1], radial[0]]) * avoidance_dir

            # Distance error from desired radius
            radius_error = dist_to_obstacle - desired_distance

            # Smooth blending between radial and tangential
            blend_factor = np.exp(-2.0 * (radius_error / desired_distance) ** 2)
            avoidance_vel = (
                radial * np.sign(radius_error) * (1 - blend_factor) * 0.3 +
                tangent * blend_factor
            )
            avoidance_vel = avoidance_vel / (np.linalg.norm(avoidance_vel) + 1e-6)
        else:
            avoidance_vel = np.array([0., 0.])

        # Combine velocities with smooth transition
        velocity = (1 - influence) * direct_vel + influence * avoidance_vel
        velocity = velocity / (np.linalg.norm(velocity) + 1e-6)
    # ...
```

Cette méthode génère un chemin qui contourne l'obstacle de manière fluide en utilisant une combinaison pondérée de composantes directes (vers la cible) et tangentielles (autour de l'obstacle). La pondération est basée sur la distance à l'obstacle et utilise des fonctions de transition douces (sigmoid et exponentielle) pour éviter les changements brusques de direction.

2.3.3 Affinement du Chemin

L'affinement du chemin implique l'identification des points du chemin actuel qui sont affectés par les obstacles. Cette opération est réalisée dans la méthode `generate_combined_targetset` :

```
def generate_combined_targetset(self, vehiclePose: CoordinatesSet, sevilleMap: list, obstacles:
↪  list) -> list:
    modified_path = sevilleMap.copy()

    for obstacle in obstacles:
        # Find all points within influence radius
        obstacle_pos = CoordinatesSet(obstacle.x, obstacle.y)
        influence_radius = obstacle.get_influence_radius()

        # Find the range of points to replace
        points_in_radius = [[]]
        emptyPointSet = True
        j = 0
        for i, point in enumerate(modified_path):
            if self.distance(point, obstacle_pos) <= influence_radius * 5:  # 5x for safety margin
                points_in_radius[j].append(i)
                emptyPointSet = False
            elif not emptyPointSet:
                j += 1
                points_in_radius.append([])
                emptyPointSet = True
```

Cette fonction identifie tous les ensembles de points consécutifs qui sont à l'intérieur du rayon d'influence de chaque obstacle. Elle utilise ensuite une stratégie d'optimisation pour ne traiter que l'ensemble de points le plus proche de l'obstacle :

```
# only select the closest pointset
# Get the pointset that has the min of the mean distance to the obstacle
min_mean_distance = float('inf')
min_mean_distance_idx = 0
for i, pointset in enumerate(points_in_radius):
    mean_distance = np.mean([self.distance(modified_path[idx], obstacle_pos) for idx in pointset])
    if mean_distance < min_mean_distance:
        min_mean_distance = mean_distance
        min_mean_distance_idx = i

points_in_radius = points_in_radius[min_mean_distance_idx]
```

Cette optimisation permet d'éviter de créer un chemin d'évitement qui croiserait la trajectoire de retour du véhicule. L'hypothèse sous-jacente étant que la probabilité que l'obstacle soit toujours présent au retour est faible, par exemple si un piéton a traversé la route.

Un nouveau chemin alternatif est ensuite généré pour éviter ces obstacles tout en assurant une transition en douceur. Ce chemin est vérifié pour s'assurer qu'il respecte les limites de la route, en utilisant des fonctions comme `check_path_within_boundaries()`.

2.3.4 Périodes de transition

Afin d'éviter des changements brusques de trajectoire, des périodes de transition sont introduites pour permettre au véhicule de s'adapter progressivement à l'évitement. L'algorithme utilise plusieurs techniques de lissage :

1. **Transition basée sur la distance** : L'influence de l'évitement augmente graduellement à mesure que le véhicule s'approche de l'obstacle :

```
transition_factor = np.clip((transition_start - dist_to_obstacle) / (transition_start -
↪  desired_distance), 0, 1)
transition_factor = np.sin(transition_factor * np.pi / 2)  # Smooth sigmoid-like curve
```

2. **Pondération exponentielle** : Une fonction gaussienne est utilisée pour moduler l'influence de l'évitement en fonction de la distance relative à l'obstacle :

```
influence = transition_factor * np.exp(-2.0 * (dist_ratio - 1.0) ** 2)
```

3. **Mélange de comportements radial et tangentiel** : Près du rayon d'influence, le véhicule privilégie un comportement tangentiel (contournement), tandis qu'à distance, il privilégie un comportement radial (éloignement) :

```
blend_factor = np.exp(-2.0 * (radius_error / desired_distance) ** 2)
avoidance_vel = (
    radial * np.sign(radius_error) * (1 - blend_factor) * 0.3 +
    tangent * blend_factor
)
```

Ces techniques de transition permettent au véhicule de naviguer de manière fluide et naturelle autour des obstacles, tout en maintenant une trajectoire globale cohérente vers sa destination. La combinaison de

fonctions trigonométriques et exponentielles crée des transitions douces qui ressemblent davantage à celles qu'un conducteur humain expérimenté effectuerait.

Pour illustrer cette approche mathématiquement, la transition peut être modélisée comme une fonction sigmoidale modifiée :

$$T(d) = \sin\left(\frac{\pi}{2} \cdot \text{clip}\left(\frac{d_{\text{start}} - d}{d_{\text{start}} - d_{\text{desired}}}, 0, 1\right)\right)$$

où d est la distance à l'obstacle, d_{start} est la distance à laquelle la transition commence, et d_{desired} est la distance cible pour l'évitement. Cette fonction produit une transition douce qui commence lentement, s'écarte rapidement de l'obstacle au milieu, puis ralentit à nouveau à l'approche de la distance cible, créant ainsi un comportement naturel et prévisible.

3 Tests et Résultats

3.1 Environnement de Simulation

Les tests initiaux du module de planification ont été réalisés dans un environnement de simulation pour valider les algorithmes avant leur déploiement sur le véhicule. Cette phase a permis d'identifier et de corriger les problèmes potentiels sans risque pour le matériel.

3.1.1 Configuration de la Simulation

La simulation a été construite autour des éléments suivants :

- Utilisation d'un fichier ROSbag pour reproduire les conditions réelles
  - Simulation des données de localisation (position ENU du véhicule)
  - Publication de la carte de référence (points de Séville)
  - Injection d'obstacles virtuels à des positions prédéfinies

Cette approche a permis de tester le module dans des conditions variées, en modifiant les paramètres des obstacles (position, taille, nombre) pour évaluer la robustesse du système.

3.1.2 Visualisation et Analyse

Le module `debug_plot` a fourni une interface visuelle essentielle pour l'analyse des performances du système :

- Affichage en temps réel de la trajectoire du véhicule
- Représentation des obstacles et de leurs zones d'influence
- Visualisation des chemins d'évitement générés
- Indicateurs de déviation latérale et de respect des limites de voie

Ces visualisations ont permis d'identifier rapidement les comportements inattendus et d'affiner les paramètres des algorithmes. Par exemple, l'ajustement du facteur d'influence des obstacles a été optimisé pour équilibrer sécurité et efficacité du parcours.

3.1.3 Résultats des Tests de Simulation

Les simulations ont démontré que le module pouvait générer des trajectoires d'évitement efficaces tout en maintenant le véhicule dans les limites de sa voie. Les métriques clés suivantes ont été évaluées :

- Distance minimale aux obstacles : maintenue au-dessus du seuil de sécurité de 2 mètres
- Respect des limites de voie : 100% des trajectoires générées restaient dans les limites définies
- Continuité des trajectoires : transitions fluides entre les segments normaux et d'évitement

Un exemple de test est illustré dans la figure suivante, montrant l'évitement d'un obstacle statique placé sur la trajectoire du véhicule. On observe comment le système génère une trajectoire alternative (en bleu) qui contourne l'obstacle tout en restant dans les limites de la voie.

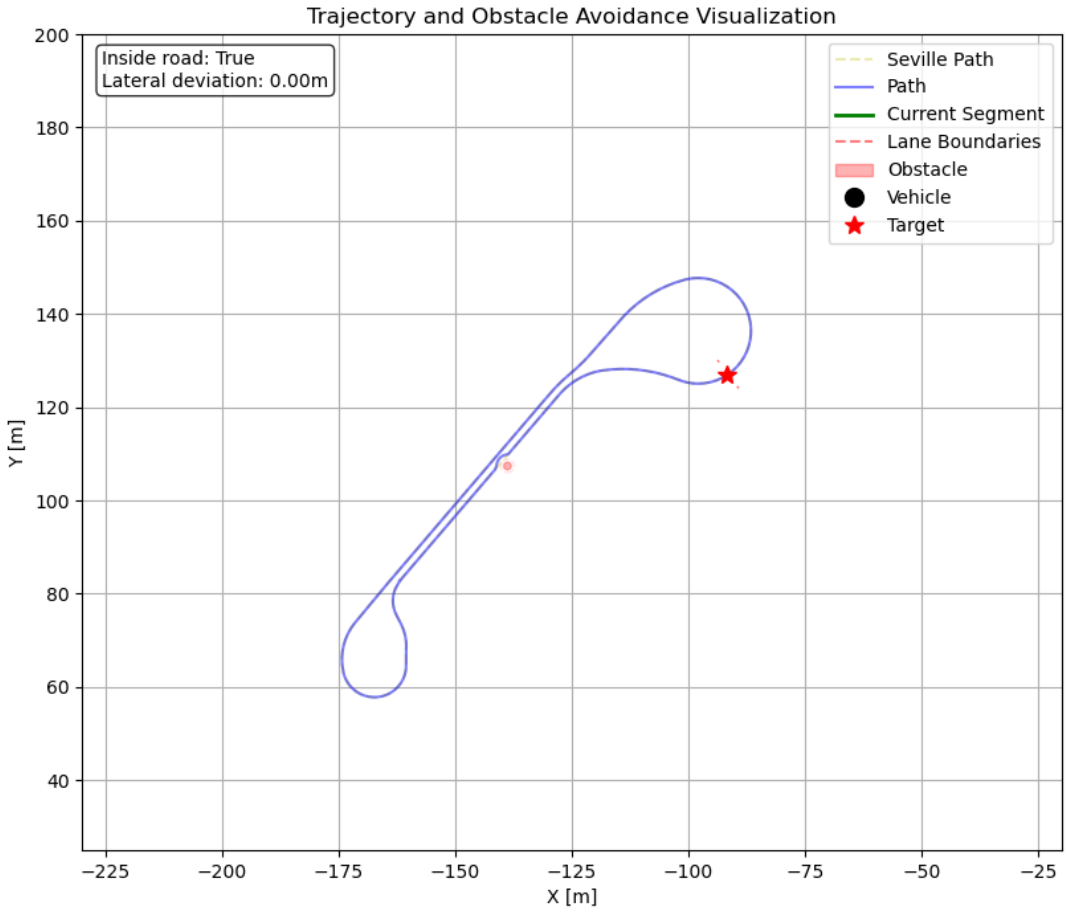

Figure 6: Trajectoire avec l'évitement d'un obstacle

3.2 Essais sur Véhicule Réel

Suite aux validations en simulation, le module a été déployé sur la Renault Zoé pour des tests en conditions réelles.

3.2.1 Intégration avec le Véhicule

L'intégration a nécessité plusieurs adaptations :

- Configuration du réseau ROS pour assurer la communication entre les différents modules
- Calibration des paramètres pour correspondre aux caractéristiques réelles du véhicule
- Synchronisation avec les modules développés par les autres groupes de l'UV SY27 (perception, localisation)

Une caractéristique notable du module est son autonomie relative : il dépend uniquement de la localisation et de la perception, le contrôle étant réalisé par un module séparé développé par Cyril PETER dans le cadre de la même TX.

3.2.2 Résultats des Tests sur Route

Les essais sur route ont été documentés dans une vidéo disponible à l'adresse : https://youtu.be/ri2ywugLCno

Ces tests ont confirmé les capacités du système en conditions réelles, mais ont également révélé certaines différences par rapport aux simulations :

- Le système a correctement suivi la trajectoire de référence dans différentes conditions
- L'évitement d'obstacles a fonctionné comme prévu, maintenant les distances de sécurité, cependant, les transitions étaient plus brusques que prévu, malgrès l'approche en sigmoïde
- La réactivité du système s'est avérée suffisante pour répondre aux changements dynamiques de l'environnement

3.2.3 Problèmes Rencontrés et Solutions

La principale différence entre simulation et réalité concernait la dynamique du véhicule :

- **Problème** : Accoups de volant lors des changements de direction
  - Le contrôleur de volant supposait un temps de réponse nul, alors que le véhicule réel présente une latence mécanique
- **Solutions proposées** :
  - Au niveau du module de décision : Fournir une avance de phase au contrôleur en transmettant des points cibles plus éloignés sur la trajectoire
  - Au niveau du contrôle : Ajouter un filtre passe-bas pour lisser les commandes de direction

4 Conclusion

Ce travail de développement et d'intégration d'un module de planification et de décision pour véhicule autonome s'inscrit dans le cadre de la TX7676, avec des composants réutilisés par l'UV SY27/AI36.

Le module développé représente une avancée significative pour le projet UTonome. L'architecture modulaire conçue permet une navigation efficace et sécurisée dans des environnements dynamiques, avec des capacités d'évitement d'obstacles et de planification de trajectoires. La migration de MATLAB vers Python a permis de résoudre les problèmes d'incompatibilité avec le système ROS du véhicule, tout en conservant les avantages analytiques de MATLAB grâce à l'interopérabilité entre les deux environnements.

Une attention particulière a été portée à la maintenabilité du code pour les futures itérations du projet UTonome. La structure modulaire, la documentation approfondie et l'utilisation de pratiques de programmation modernes comme les dataclasses et le typage statique facilitent la compréhension et l'extension du code par les futurs membres de l'association. Les algorithmes d'évitement d'obstacles et de planification de trajectoires ont été conçus pour être facilement paramétrables et adaptables à différentes configurations de véhicules et d'environnements.

Les essais d'intégration Matlab sur le véhicule ont été rendus possibles grâce à l'accès aux véhicules dans le cadre du programme SY27/AI36. Cette phase d'expérimentation, bien que relevant principalement de la TX, a permis d'identifier les incompatibilités critiques entre les versions de ROS qui ont guidé la réorientation du projet vers une implémentation Python.

Les résultats obtenus lors des tests sur simulateur démontrent la viabilité de l'approche adoptée. Le module est capable de générer des trajectoires fluides et de contourner efficacement les obstacles tout en respectant les contraintes de la route.

Ce travail constitue une base solide pour le développement continu du projet UTonome et contribue à l'avancement des technologies de conduite autonome au sein de l'UTC. La combinaison d'approches théoriques rigoureuses et de simulations exhaustives a permis de développer un module à la fois performant et réaliste, capable de répondre aux défis complexes de la navigation autonome en environnement dynamique.