Options Robotique et Diagnostiques

Document Technique - Projet 2025/2026

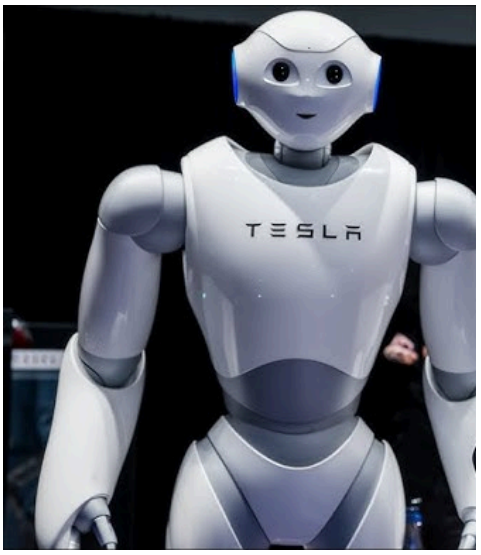
“Vigilis” : Robot de vigilance

Demandeur : Mme Seddiki

1. Introduction

a. Etat de l'art : Robotique mobile et vision par ordinateur

La robotique mobile autonome constitue aujourd'hui un domaine de recherche majeur, combinant perception, prise de décision et action dans des environnements complexes et dynamiques. Les robots mobiles sont de plus en plus utilisés dans des domaines variés tels que la surveillance, l'exploration, la logistique, l'assistance ou encore l'industrie. L'autonomie d'un robot repose principalement sur sa capacité à percevoir son environnement, à interpréter les informations reçues et à agir en conséquence.



a.1. Robots mobiles autonomes

Un robot mobile autonome est un système capable de se déplacer sans intervention humaine directe, en s'appuyant sur des capteurs pour percevoir son environnement. Ces capteurs peuvent inclure des caméras, des capteurs LiDAR, des capteurs ultrasoniques ou des centrales inertielles. Le choix des capteurs dépend fortement des contraintes du projet, telles que le coût, la complexité, la précision requise et les conditions environnementales.

Les robots mobiles sont généralement structurés autour d'une architecture modulaire, où chaque module assure une fonction spécifique : perception, localisation, planification et contrôle. Cette approche permet une meilleure évolutivité et une maintenance plus aisée du système.

a.2. Vision par ordinateur en robotique

La vision par ordinateur joue un rôle fondamental dans la robotique moderne. Elle permet au robot d'analyser des images issues de caméras afin d'identifier des objets, des obstacles ou des cibles d'intérêt. Contrairement aux capteurs de distance, la vision offre une richesse d'information élevée, notamment en termes de couleur, de forme et de texture.

Dans de nombreux projets robotiques, la vision est utilisée pour la reconnaissance d'objets, le suivi de cibles ou la navigation visuelle. Cependant, elle reste sensible aux conditions d'éclairage et aux variations environnementales.

a.3. Méthodes classiques de détection d'objets

Les méthodes classiques de vision par ordinateur reposent principalement sur des techniques de traitement d'image telles que le seuillage, la segmentation par couleur ou la détection de contours. L'espace colorimétrique HSV est souvent privilégié car il sépare la teinte de la luminosité, ce qui rend la détection des couleurs plus robuste face aux variations d'éclairage.

Ces méthodes présentent l'avantage d'être simples à implémenter, rapides à exécuter et peu coûteuses en ressources de calcul. Elles sont particulièrement adaptées aux systèmes embarqués ou aux environnements simulés.

a.4. Méthodes avancées et apprentissage profond

Les approches modernes reposent de plus en plus sur l'apprentissage profond, notamment à travers des réseaux de neurones convolutionnels (CNN) comme YOLO ou Faster R-CNN. Ces méthodes offrent une grande robustesse et une capacité de généralisation élevée, mais nécessitent des ressources de calcul importantes et des jeux de données annotés.

b. Contexte du Master 1 et objectifs du projet

Dans le cadre du Master 1 Informatique et Big Data, l'unité d'enseignement **Robotique et aide au diagnostic**, dirigée par Madame Seddiki, nous a offert une dimension concrète dans le monde de la robotique mobile. Ce domaine, à la limite entre de l'informatique logicielle et de l'électronique, repose aujourd'hui sur des standards industriels tels que *ROS (Robot Operating System).*

Le projet qui nous a été confié consiste en une première initiation pratique visant à comprendre les mécanismes de perception et d'action d'un robot autonome et la prise en main d'un logiciel comme ROS. Nous avons choisi de développer *"Vigilis"*, un robot spécialisé dans la reconnaissance de menace.

c. Présentation du projet VIGILIS

Le projet Vigilis à pour but un scénario de surveillance et de diagnostic d'environnement. L'objectif est de concevoir un système capable d'identifier de manière autonome un élément perturbateur ou une "menace" au sein d'un espace donné. Dans le cadre de cette initiation, la menace est représentée par un objet, un cylindre de couleur rouge.

Le robot doit être capable de :

- Scanner son environnement via un flux vidéo en temps réel.
- Diagnostiquer la présence d'une cible en isolant sa signature colorimétrique.
- Adapter son comportement moteur pour suivre cette cible à une distance de sécurité constante.

d. Problématique

La problématique centrale de notre étude est la suivante :

“*Comment peut-on programmer le robot Vigilis pour qu’il identifie une cible par sa couleur et parvienne à la suivre tout seul ?*”

Cela implique de répondre à plusieurs défis techniques :

- L'interprétation des données de la caméra via des algorithmes de vision par ordinateur.
- La gestion de la communication entre les différents processus du système.
- La régulation de la vitesse et de la trajectoire pour maintenir un asservissement visuel fluide.

2. Environnement Technique et Configuration

Cette section détaille les outils et l'infrastructure logicielle utilisés pour le développement du projet Vigilis. La compréhension de l'écosystème ROS et la maîtrise de la simulation sont des prérequis essentiels à toute mission d'aide au diagnostic.

a. Écosystème ROS Noetic : Concepts de base

Le *Robot Operating System (ROS)*, dans sa version Noetic, a servi de colonne vertébrale à notre projet. ROS facilite la communication entre les différents composants logiciels du robot. Nous avons articulé notre travail autour de trois concepts fondamentaux :

- **Les Nœuds (Nodes)** : Ce sont des processus indépendants qui effectuent des calculs. Dans notre architecture, nous avons notamment développé *red_object_detector_node* pour la vision et *object_follower_node* pour le contrôle moteur.
- **Les Topics** : canaux de communication nommés par lesquels les nœuds s'échangent des informations. Par exemple, les données d'image circulent sur le topic */camera/rgb/image_raw*, tandis que les ordres de mouvement sont envoyés sur */cmd_vel*.
- **Les Messages** : Formats de données structurés circulant sur les topics. Nous avons utilisé des messages de type *Image* pour la perception et *Twist* pour la vélocité du robot.

b. Simulation avec Gazebo

Pour ce projet de "Robotique et aide au diagnostic", l'utilisation du simulateur Gazebo s'est avérée indispensable pour plusieurs raisons :

- **Sécurité et Intégrité :** Tester un algorithme de suivi de cible sur un robot réel comporte des risques de collision et est un véritable coût financier. Gazebo permet de simuler la physique réelle (gravité, friction, collisions) dans un environnement sans danger.
- **Reproductibilité :** Il est facile de réinitialiser le robot Vigilis et sa cible à des positions exactes pour comparer l'efficacité de nos ajustements de code.
- **Aide au diagnostic préventif :** Gazebo permet de visualiser les capteurs en temps réel, facilitant ainsi le diagnostic de bugs logiciels avant tout déploiement physique.

Nous avons aussi la possibilité de créer un monde depuis un monde vide depuis Gazebo, ce que nous avons fait et cela faisait partie des objectifs pédagogiques demandés.Nous pouvions changer les caractéristiques de ce monde via le fichier d'extension *.world*. C'est notamment comme cela que nous avons pu changer les cylindres de couleur sur la partie <*ambiant*> et <*diffuse*> du code.

c. Infrastructure de l'équipe : WSL vs Machine Virtuelle

En tant que binôme, nous avons opté pour deux méthodes d'installation différentes sur **Ubuntu 20.04.6.**

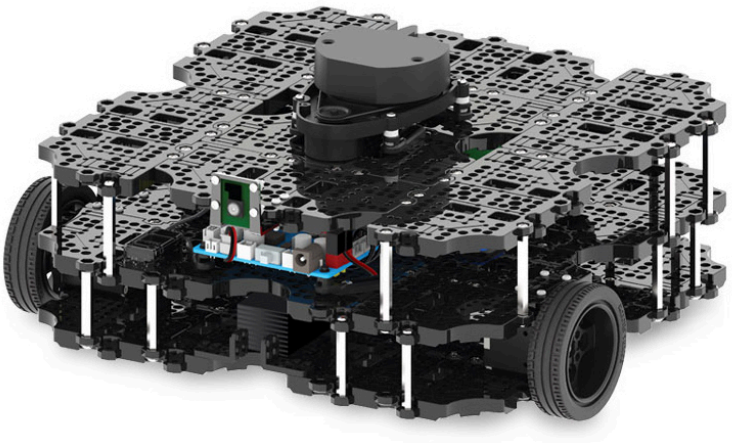
Critère	**Environnement de Chloé**	**Environnement de Tugba**
Technologie	WSL (Windows Subsystem for Linux)	Machine Virtuelle (VMware/VirtualBox)
Avantages	Performances quasi-natives, accès direct aux ressources CPU.	Isolation complète du système, gestion simplifiée de l'affichage.
Contraintes	Nécessite une interface graphique (serveur X) pour visualiser Gazebo.	Consommation plus élevée de ressources RAM.

Nous avons cependant décidé de travailler sur la machine de Tugba car elle était plus robuste et "stable".

d. Le robot TurtleBot3 Waffle : Caractéristiques et avantages

Pour le projet Vigilis, nous avons configuré le modèle waffle. Bien que le modèle Burger soit plus compact, le choix du Waffle était impératif pour notre problématique d'identification de cible :

- **Capteur de vision :** Le Waffle est équipé nativement d'une caméra, élément absent du modèle Burger qui ne possède qu'un LiDAR. Sans cette caméra, la détection par couleur via OpenCV serait impossible.



3. Architecture du Workspace et Outils Logiciels

Pour garantir la fluidité du développement et la modularité du code, nous avons structuré notre environnement de travail selon les standards de ROS Noetic.

a. Structure du Workspace *catkin_ws_3*

Nous avons créé un espace de travail dédié nommé *catkin_ws_3* comme on a fait lors des TPs. L'organisation suit une hiérarchie précise permettant à l'outil *catkin_make* de compiler efficacement les dépendances et de générer les exécutables nécessaires.

- **/catkin_ws_3** : Dossier racine de notre environnement.
- **/src** : Répertoire contenant l'ensemble du code source.
- **/robot** : Notre package ROS personnalisé, créé pour cette mission.
- **/scripts** : Ce sous-dossier contient nos deux scripts Python principaux (*object_detector.py* et *object_follower.py*).

Cette structure permet de séparer proprement la configuration du robot, les environnements (Worlds) et la logique algorithmique (Scripts). Le passage par la commande *catkin_make* à la racine du workspace assure que nos nœuds sont reconnus par l'écosystème ROS et peuvent être lancés depuis n'importe quel terminal.

b. Bibliothèques utilisées : OpenCV et CV Bridge

Le diagnostic visuel repose sur le traitement d'image haute performance. Pour cela, deux bibliothèques ont été essentielles :

- **OpenCV (cv2)** : Bibliothèque leader pour la vision par ordinateur. Nous l'avons utilisée pour transformer le flux vidéo brut en une "carte de menace" via le filtrage de couleurs, la détection de contours et le calcul de moments géométriques.
- **CV Bridge** : Dans ROS, les images circulent sous forme de messages *sensor_msgs/Image*. Cependant, OpenCV traite des matrices NumPy. CvBridge joue le rôle de traducteur indispensable, permettant de convertir les messages ROS en formats compatibles OpenCV pour le traitement, puis de les renvoyer si nécessaire.

c. Le fichier de lancement (*.launch*) : Automatisation

Plutôt que de lancer chaque nœud manuellement dans des terminaux séparés comme en TP, nous avons conçu un fichier d'automatisation nommé *projet.launch*. Ce fichier est le chef d'orchestre de la simulation. Ça montre également notre capacité à faire du code modulaire.

Il réalise les actions suivantes en une seule commande :

- **Chargement du modèle** : Il définit le TurtleBot3 Waffle comme modèle par défaut.
- **Initialisation du monde** : Il lance Gazebo avec notre environnement personnalisé *projet.world*.
- **Spawn du Robot** : Il positionne le robot dans la simulation aux coordonnées (0, 0, 0.05).
- **Exécution des Nodes** : Il démarre simultanément *red_object_detector_node* pour la détection et *object_follower_node* pour le suivi.

L'utilisation du paramètre *output="screen"* dans le fichier *.launch* nous permet de suivre les logs et le diagnostic du robot (ex: "Cible perdue", "Suivi actif") directement dans la console principale.

4. Modélisation fonctionnelle du système Vigilis

Le système Vigilis repose sur une architecture fonctionnelle claire, divisée en plusieurs modules coopérant pour assurer le comportement autonome du robot.

a. Architecture globale

Le fonctionnement du système peut être décrit selon une boucle perception-décision-action. La caméra embarquée fournit des images RGB qui sont traitées par le module de perception. Les informations extraites sont ensuite transmises au module de décision, qui génère des commandes de mouvement envoyées au robot.

b. Module de perception

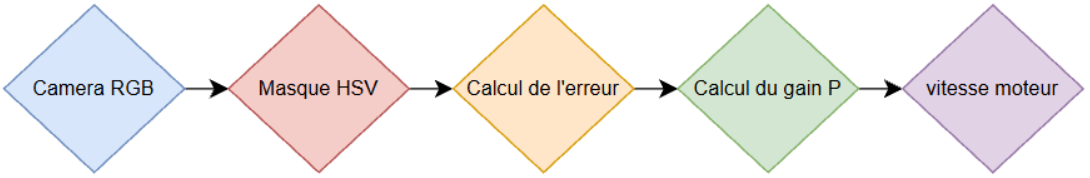
Le module de perception est chargé de détecter les objets rouges présents dans l'environnement. Il applique une segmentation par couleur dans l'espace HSV afin d'isoler les régions correspondant à la couleur rouge. Pour chaque région détectée, une aire est calculée, ce qui permet d'estimer la distance relative de l'objet par rapport au robot. Afin de répondre à l'objectif du projet, le système sélectionne l'objet rouge le plus éloigné, correspondant à la plus petite aire détectée.

c. Module de décision

Le module de décision reçoit deux informations principales : l'erreur angulaire par rapport à la cible et la taille estimée de l'objet. À partir de ces données, il ajuste la vitesse linéaire et angulaire du robot à l'aide d'un contrôleur proportionnel.

d. Module d'action

Le module d'action applique les commandes générées au robot via le topic */cmd_vel*. Le robot adapte ainsi son orientation et sa vitesse pour se diriger vers la cible sélectionnée.



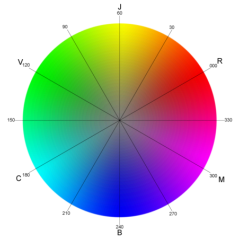
5. Perception : Le Nœud de Détection

Le nœud *red_object_detector_node* a pour mission d'analyser les images provenant de la caméra du TurtleBot3 Waffle pour localiser une cible rouge. Cette étape de perception est la première phase du cycle « Perception-Décision-Action » de notre robot

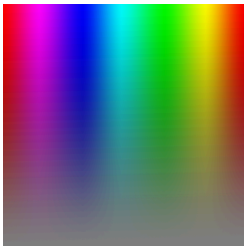
a. Prétraitement de l'image : Espace colorimétrique HSV vs BGR

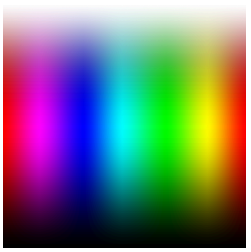
Par défaut, la caméra du robot capture des images au format **BGR** (Bleu, Vert, Rouge). Cependant, ce format est très sensible aux variations de luminosité, ce qui peut fausser le diagnostic. Nous avons donc utilisé OpenCV pour convertir l'image dans l'espace **HSV** (Hue, Saturation, Value).

L'espace **HSV** (**H**ue, **S**aturation, **V**alue) est une manière de représenter les couleurs en se rapprochant de la perception humaine, contrairement au format **RGB** (Rouge, Vert, Bleu) qui est une simple combinaison de lumières.



1. **Hue (Teinte - H) :** C'est la "couleur" proprement dite (rouge, orange, jaune, vert, bleu, etc.). Elle est représentée sur un cercle de 0 à 360°. Dans OpenCV, pour que cela tienne sur un octet, la valeur est divisée par 2. On utilise donc une échelle de **0 à 180**.
2. **Saturation (S) :** C'est l'intensité ou la pureté de la couleur.
    - 0% : La couleur est grise (très délavée).
    - 100% : La couleur est très vive et riche.
3. **Value (Valeur / Luminosité - V) :** C'est la brillance de la couleur.
    - 0% : C'est le noir total.
    - 100% : La couleur est à son maximum de luminosité.





Pourquoi utiliser le HSV plutôt que le RGB en robotique ?

- **En RGB :** Si une ombre passe sur votre objet rouge, les trois valeurs (R, G, B) changent drastiquement. Le robot peut alors "perdre" la cible car il ne reconnaît plus les chiffres.
- **En HSV :** Si la lumière change, seule la **Value (V)** va varier. La **Teinte (H)** restera globalement la même. Cela permet à Vigilis de garder son objectif sur la cible même si l'éclairage dans Gazebo n'est pas parfait.

b. Segmentation : Filtrage de la couleur rouge (les seuils de menace)

La détection du rouge présente une particularité dans OpenCV : cette couleur se situe aux deux extrémités du spectre des teintes (proche de 0° et de 180°). Pour assurer une détection robuste de la menace, notre code combine deux masques de filtrage:

- **Plage 1 :** Teintes de 0 à 10.
- **Plage 2 :** Teintes de 170 à 180.

En additionnant ces deux masques, nous obtenons une image binaire où seuls les objets rouges apparaissent en blanc sur fond noir. Pour éviter de détecter des bruits parasites (petits points rouges isolés), nous avons implémenté un seuil de surface minimale via *cv2.contourArea*.

c. Analyse d'image : Extraction du centre de masse (Centroid) et calcul de l'erreur

Une fois la menace isolée, le script calcule les moments géométriques du contour pour déterminer son centre de masse (le centroïde C_x). L'objectif est que le robot garde la cible au centre de son champ de vision. L'erreur de positionnement, notée E , est calculée par la formule suivante :

$$E = C_x - \left(\frac{Largeur_{image}}{2}\right)$$

- Si $E > 0$: L'objet est à droite du centre.
- Si $E < 0$: L'objet est à gauche du centre.
- Si $E = 0$: L'objet est parfaitement centré.

d. Publication des données de diagnostic

Plutôt que de tout traiter dans un seul script, Vigilis publie ses résultats sur deux topics distincts:

- */object_error* (Float32) : Transmet le décalage horizontal pour corriger la trajectoire.
- */object_size* (Float32) : Transmet la surface de l'objet, ce qui sert d'indicateur de distance (plus l'objet est gros, plus il est proche).

Cette séparation permet à d'autres nœuds de s'abonner à ces données sans avoir à refaire le calcul de vision $E \approx 0$.

6. Commande : Le Nœud de Suivi

Le nœud *object_follower_node* est responsable de la génération des commandes motrices. Il implémente une logique d'asservissement visuel qui permet au TurtleBot3 Waffle de maintenir la cible dans son axe tout en gérant sa distance de sécurité.

a. Stratégie d'asservissement visuel

Le robot utilise une boucle de rétroaction simple mais efficace. Il s'abonne aux topics */object_error* et */object_size* pour recevoir les informations traitées par le nœud de vision. L'objectif est de minimiser l'erreur angulaire pour que la cible reste bien au centre de l'image $E \approx 0$.

b. Algorithme de contrôle : Lissage et Gain Proportionnel

Pour éviter les mouvements brusques et saccadés qui pourraient faire perdre la cible de vue, nous avons implémenté deux mécanismes clés :

- **Le Lissage (Filtre Passe-bas)** : Nous utilisons une moyenne glissante avec un coefficient α = 0.2 pour lisser les variations de l'erreur angulaire. Cela permet de filtrer les bruits de détection.
- **Formule** : $Erreur_{lisse} = (\alpha * Erreur_{actuelle}) + (1 - \alpha) * Erreur_{précédente}$
- **Le Contrôle Proportionnel (P)** : La vitesse de rotation est proportionnelle à l'erreur détectée via un gain $K_p = 0.0003$. Plus l'objet est loin du centre, plus le robot tourne vite pour se recadrer.

c. Logique de décision : Recherche de cible, Suivi actif et Arrêt de sécurité

Le comportement de Vigilis est régi par trois états basés sur la taille de l'objet (indicateur de distance):

- **État de Recherche** : Si la taille de l'objet est inférieure à 300 pixels (cible perdue ou trop loin), le robot s'arrête de progresser et tourne sur lui-même à une vitesse de $0.2\ rad/s$ pour scanner son environnement.
- **État de Suivi Actif** : Si l'objet est détecté, le robot avance à une vitesse linéaire constante de $0.1\ ms/s$ tout en ajustant sa trajectoire angulaire pour rester centré.
- **État d'arrêt de Sécurité** : Dès que l'objet atteint une taille critique 650 000 pixels dans notre code), le robot considère qu'il est à proximité immédiate de la menace et stoppe tout mouvement pour éviter une collision.

d. Communication avec les moteurs

Pour piloter les moteurs du Waffle, le nœud publie des messages sur le topic */cmd_vel*.

- ***linear.x*** : Définit la vitesse d'avance en ms.
- ***angular.z*** : Définit la vitesse de rotation rad/s.

Cette structure standardisée permet à notre code d'être parfaitement compatible avec n'importe quel robot supportant ROS, garantissant alors la portabilité de notre solution de diagnostic.

7. Tests, Analyse et Résultats

La validation du projet **Vigilis** s'est déroulée dans l'environnement de simulation Gazebo, permettant de confronter nos algorithmes à des conditions physiques réalistes (inertie, collisions, champ de vision).

a. Simulation du scénario dans Gazebo

Pour valider notre problématique, nous avons placé un objet cylindrique rouge dans un monde Gazebo vide (chargé via *project.launch*). Le test s'est décomposé en trois phases observées en temps réel :

- **Phase de recherche :** Au démarrage, sans cible en vue, le robot Vigilis a entamé une rotation sur lui-même à 0.2 rad/s. Nous avons vérifié via le terminal que le message *"FOLLOWER: Cible perdue, recherche..."* s'affichait bien.

- **Phase d'Interception :** Dès que l'objet rouge est entré dans le champ de vision de la caméra du Waffle, le nœud de détection a isolé le masque et calculé l'erreur de centrage. Le robot a alors cessé sa rotation pour avancer à 0.1 rad/s tout en ajustant son angle.

- **Phase de Sécurité** : Une fois le robot arrivé à proximité de la cible (surface détectée $\geq$ 650,000 pixels), le nœud de suivi a ordonné un arrêt complet (*linear.x = 0.0* et *angular.z = 0.0*).

b. Analyse des performances : Précision du suivi et fluidité

L'utilisation d'un filtre passe-bas ($\alpha = 0.2$) sur l'erreur angulaire a grandement amélioré la stabilité du robot. Sans ce lissage, nous avions observé lors de nos premiers essais que le robot "oscillait" violemment de gauche à droite, risquant de perdre la cible à cause du flou de mouvement simulé.

Le gain proportionnel K_p = 0.0003 s'est avéré être un excellent compromis:

- Il est assez élevé pour que le robot réagit promptement aux déplacements de la cible.
- Il est assez faible pour éviter des rotations brusques qui dépasseraient les capacités physiques du moteur simulé.

c. Difficultés rencontrées et solutions apportées (ex: instabilité de l'image, latence)

Problème	Cause Identifiée	Solution Apportée
Latence de réaction	Le traitement d'image est gourmand en CPU, surtout en VM.	Optimisation du code et réduction de la taille de la fenêtre d'affichage (*cv2.resize* à 320 x 240) pour fluidifier le traitement.
Installation de ROS et Gazebo	Conflits de versions et dépendances manquantes lors de l'installation des outils	Vérification de la compatibilité des versions et installation progressive des dépendances à l'aide de la documentation officielle
Gestion du monde Gazebo	Paramétrage complexe de l'environnement (objets, camera) impactant la simulation. Changer la couleur des objets…	Ajustement du monde Gazebo et tests pour obtenir un environnement stable et cohérent
Développement du follower	Difficultés à exploiter correctement les données issues de la vision pour le suivi de la cible. Bien régler les variables pour que le robot bouge bien…	Séparation des traitements en plusieurs topics et réglage progressif des seuils et paramètres.
Programmation des déplacements du robot	Mauvaise synchronisation entre perception et commandes de mouvement	Mise en place de plusieurs tests.
Intégration d'un capteur LiDAR	Le traitement des données 3D (nuages de points) est extrêmement gourmand en ressources CPU/RAM, ce qui provoque une instabilité de la machine virtuelle (VM).	Bien que le capteur LiDAR n'ait pas été activé dans cette phase de simulation pour optimiser les ressources CPU/RAM, son intégration restera une étape clé. Cela sera rendu possible grâce à une optimisation future du matériel ou lors de l'utilisation sur un robot physique réel.

8. Conclusion et Perspectives

Le projet **Vigilis** a constitué une étape majeure dans notre apprentissage de la robotique mobile. En partant d'une page blanche avec la pratique des TPs, nous avons réussi à concevoir un système complet capable de percevoir son environnement, de diagnostiquer une menace colorée et “d'agir” en conséquence.

a. Bilan sur l'initiation à la robotique et ROS

Cette expérience nous a permis de passer de la théorie vue en cours à une application pratique concrète. La modularité de ROS, illustrée par la séparation entre le nœud de détection (*object_detector.py*) et le nœud de suivi (*object_follower.py*), nous a appris l'importance d'une architecture logicielle décentralisée.

L'utilisation conjointe de Gazebo et de Noetic via le fichier *projet.launch* a démontré l'efficacité de la simulation pour le prototypage rapide. Nous avons appris à gérer les flux de données asynchrones (*Topics*) et à transformer des données brutes de capteurs en commandes motrices réelles.

À travers le scénario de suivi de cible rouge, nous avons exploré les prémices de l'aide au diagnostic. Pour un robot, "diagnostiquer" signifie interpréter des données hétérogènes pour qualifier une situation. Dans notre cas, Vigilis a dû :

- Diagnostiquer la présence d'une menace représentée par le cylindre rouge (Via OpenCV) .
- Diagnostiquer la distance et la position relative de cette menace.
- Diagnostiquer une situation d'urgence (arrêt complet si la cible est trop proche).

Ce projet montre qu'en couplant la robotique au Big Data, nous pourrions stocker ces données de diagnostic sur de longues périodes pour analyser des tendances, détecter des anomalies récurrentes ou entraîner des modèles de prédiction plus complexes.

b. Évolutions possibles : Vers un diagnostic multi-menaces

Bien que fonctionnel, le système Vigilis dispose d'une marge de progression importante que nous pourrions explorer dans des travaux futurs :

- **Multi-diagnostic** : Entraîner le robot à reconnaître plusieurs types de menaces avec des comportements différents (ex: fuir devant une couleur bleue, suivre la couleur rouge). Peut-être avec un réseau de neurones comme YOLO. Remplacer ou compléter la détection par seuils de couleur?
- **Mettre un LiDAR** : Intégrer les données du LiDAR du Waffle pour que le robot puisse situer les menaces sur une carte de son environnement, alliant ainsi vision et navigation spatiale et pouvant éviter les obstacles gênants.

En conclusion, ce projet sous la direction de Madame Seddiki a parfaitement rempli son rôle d'initiation. Il nous a donné les clés techniques et méthodologiques pour aborder avec confiance des systèmes robotiques plus complexes et les intégrer dans des architectures de données à grande échelle. Nous avons surtout eu une autre vision de la robotique.

9. Sources & Bibliographie

Cette section répertorie les ressources documentaires, les outils logiciels et les bibliothèques techniques qui ont permis la réalisation et la validation du projet **Vigilis**.

a. Documentation officielles

[1] **ROS Wiki – ROS Noetic Ninjemys**
Documentation officielle de ROS décrivant l'architecture des nœuds, la communication par messages (topics), la gestion des services et les outils de diagnostic (roscore, roslaunch, rqt_graph).
Disponible à l'adresse : http://wiki.ros.org/noetic

[2] **OpenCV Documentation**
Référence technique du framework OpenCV utilisée pour le traitement d'images, la conversion des espaces colorimétriques (BGR -> HSV) et les techniques de segmentation par seuillage.
Disponible à l'adresse : https://docs.opencv.org/

[3] **Robotis – TurtleBot3 e-Manual**
Manuel officiel du TurtleBot3 décrivant les caractéristiques matérielles du modèle Waffle, la configuration des capteurs (caméra, LiDAR) et le pilotage du robot via le topic `/cmd_vel`.
Disponible à l'adresse :
https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

b. Outils de développement et logiciels

[4] **Gazebo Simulator**
Environnement de simulation 3D permettant la modélisation physique du robot, l'intégration de capteurs et la visualisation du comportement du système en temps réel.
Open Source Robotics Foundation (OSRF).
Disponible à l'adresse : https://gazebosim.org/home

[5] **CvBridge (ROS Package)**
Bibliothèque ROS permettant la conversion des messages `sensor_msgs/Image` en structures compatibles avec OpenCV pour le traitement d'images en Python.
Intégré à la distribution ROS Noetic.
Disponible à l'adresse : https://wiki.ros.org/cv_bridge

[6] **Python 3**
Langage de programmation utilisé pour le développement des nœuds ROS de perception et de commande du robot Vigilis.
Python Software Foundation.
Disponible à l'adresse : https://www.python.org/downloads/

[7] **Ultralytics - YOLOv8 Object Detection Documentation**
Documentation officielle de la bibliothèque YOLO pour la détection d'objets par réseaux de neurones.
Disponible à l'adresse : https://docs.ultralytics.com/tasks/detect/

c. Références académiques

[8] **Seddiki, M.**
Robotique et Aide au Diagnostic, supports de TP, Master 1 Informatique et Big Data, Université Paris 8, 2025.

d. Concepts techniques et fondements

[9] **What is LiDAR? - Ansys**
Présentation générale de la technologie LiDAR, ses principes de fonctionnement et ses applications.
Disponible à l'adresse : https://www.ansys.com/fr-fr/simulation-topics/what-is-lidar

[10] **HSV Color Model - Coloraide**
Description du modèle de couleur HSV (Hue, Saturation, Value) et de son fonctionnement.
Disponible à l'adresse : https://facelessuser.github.io/coloraide/colors/hsv/

[11] **Introduction au filtrage des signaux - Manzaner**
Support de cours sur le filtrage, incluant des notions de moyenne glissante et filtres passe-bas.
Accessible via les supports pédagogiques de l'auteur.

[12] **Inria - Robotique**
Page institutionnelle de l'INRIA sur la robotique ; contextes et enjeux des systèmes robotiques modernes.
Disponible à l'adresse : https://www.inria.fr/fr/robotique

e. Sources audio-visuelles

[13] **YouTube - Traversaro, F.**
« How LiDAR Works - Lidar Explained » (explication visuelle des principes LiDAR), publié sur YouTube.
Disponible à l'adresse : https://www.youtube.com/watch?v=q3upozYnmec

[14] **YouTube - Learn ROS**
« ROS for Beginners: Learn ROS in 5 Days » (tutoriel de ROS), vidéo disponible sur YouTube.
Disponible à l'adresse : https://youtu.be/FQ6ahcJOVz0

[15] **YouTube - Robotique et IA**
« Humanoid Robotics and Embodied AI Explained » (analyse des systèmes robotiques et de l'IA), vidéo disponible sur YouTube.
Disponible à l'adresse : https://www.youtube.com/watch?v=Ak4S2EEVBqc

f. Articles grand public (contexte et perspectives)

[16] **IBM - Humanoid Robotics and AI**
Article expliquant l'impact et les perspectives des robots humanoïdes et de l'IA.
Disponible à l'adresse :
https://www.ibm.com/fr-fr/think/insights/humanoid-robotics-and-embodied-ai-are-about-to-change-the-world

10. Annexes

object_follower.py

Open

object_follower.py
~/catkin_ws_3/src/robot/script

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import rospy
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist
import numpy as np

class ControleurObjet:
    def __init__(self):
        rospy.init_node('object_follower_node', anonymous=True)

        rospy.Subscriber('/object_error', Float32, self.rappel_erreur)
        rospy.Subscriber('/object_size', Float32, self.rappel_taille)
        self.pub_cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=1)
        self.vitesse_cmd = Twist()

        self.Kp_angle = 0.0003
        self.taille_cible = 650000
        self.taille_min = 300

        self.erreur_angle = 0.0
        self.erreur_angle_lisse = 0.0
        self.taille_objet = 0.0

        rospy.Timer(rospy.Duration(0.1), self.controler_robot)

    def rappel_erreur(self, data):
        self.erreur_angle = data.data

    def rappel_taille(self, data):
        self.taille_objet = data.data

    def controler_robot(self, event):
        alpha = 0.2  # lissage
        self.erreur_angle_lisse = alpha * self.erreur_angle + (1 - alpha) * self.erreur_angle_lisse

        # Limite de l'erreur angulaire
        erreur_max = 1000
        if abs(self.erreur_angle_lisse) > erreur_max:
            self.erreur_angle_lisse = np.sign(self.erreur_angle_lisse) * erreur_max

        if self.taille_objet <= self.taille_min:
            self.vitesse_cmd.linear.x = 0.0
            self.vitesse_cmd.angular.z = 0.2
            rospy.logwarn("FOLLOWER: Cible perdue, recherche...")

        elif self.taille_objet >= self.taille_cible:
            self.vitesse_cmd.linear.x = 0.0
            self.vitesse_cmd.angular.z = 0.0
            rospy.loginfo("FOLLOWER: Cible atteinte, arrêt complet.")

        else:
            self.vitesse_cmd.angular.z = -self.erreur_angle_lisse * self.Kp_angle
            self.vitesse_cmd.linear.x = 0.1
            rospy.loginfo("FOLLOWER: Suivi actif, angular: %.2f", self.vitesse_cmd.angular.z)

        self.pub_cmd_vel.publish(self.vitesse_cmd)

def main():
    node = ControleurObjet()
    rospy.spin()
```

object_detector.py

Open

object_detector.py
~/catkin_ws_3/src/robot/script

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import rospy
import cv2
import numpy as np
from sensor_msgs.msg import Image
from std_msgs.msg import Float32
from cv_bridge import CvBridge, CvBridgeError

class DetecteurObjet:
    def __init__(self):
        rospy.init_node('red_object_detector_node', anonymous=True)
        self.bridge = CvBridge()

        self.largeur_image = 640
        self.seuil_aire_min = 50
        self.seuil_erreur_ignore = 2
        self.target_locked = False

        self.pub_erreur = rospy.Publisher('/object_error', Float32, queue_size=1)
        self.pub_taille = rospy.Publisher('/object_size', Float32, queue_size=1)

        rospy.Subscriber("/camera/rgb/image_raw", Image, self.rappel_image)

    def rappel_image(self, data):
        try:
            image_cv = self.bridge.imgmsg_to_cv2(data, "bgr8")
        except CvBridgeError as e:
            rospy.logerr(e)
            return

        hsv = cv2.cvtColor(image_cv, cv2.COLOR_BGR2HSV)

        bas_rouge_1 = np.array([0, 100, 100])
        haut_rouge_1 = np.array([10, 255, 255])
        bas_rouge_2 = np.array([170, 100, 100])
        haut_rouge_2 = np.array([180, 255, 255])

        masque = cv2.inRange(hsv, bas_rouge_1, haut_rouge_1) + cv2.inRange(hsv, bas_rouge_2, haut_rouge_2)
        contours, _ = cv2.findContours(masque, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        erreur = 0.0
        taille = 0.0

        if contours:
            valid_contours = [c for c in contours if cv2.contourArea(c) > self.seuil_aire_min]

            if valid_contours:
                plus_petit_contour = min(valid_contours, key=cv2.contourArea)
                aire = cv2.contourArea(plus_petit_contour)

                M = cv2.moments(plus_petit_contour)
                if M["m00"] > 0:
                    cx = int(M['m10'] / M['m00'])
                    erreur_brute = cx - (self.largeur_image / 2)

                    if abs(erreur_brute) > self.seuil_erreur_ignore:
                        erreur = erreur_brute

                    taille = aire
                    self.target_locked = True

        self.pub_erreur.publish(Float32(erreur))
        self.pub_taille.publish(Float32(taille))

        masque_petit = cv2.resize(masque, (320, 240))
        cv2.imshow("Masque Rouge", masque_petit)
        cv2.waitKey(3)
```

projet.launch

Open — projet.launch — ~/catkin_ws_3/src/robot/launch

```
<launch>
    <arg name="model" default="waffle"/>
    <param name="robot_description" command="$(find xacro)/xacro --inorder $(find turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />

    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find robot)/worlds/projet.world"/>
        <arg name="paused" value="false"/>
        <arg name="use_sim_time" value="true"/>
        <arg name="gui" value="true"/>
        <arg name="headless" value="false"/>
        <arg name="debug" value="false"/>
    </include>

    <node name="spawn_urdf"
          pkg="gazebo_ros"
          type="spawn_model"
          args="-param robot_description
                -urdf
                -model turtlebot3_waffle
                -x 0 -y 0 -z 0.05"/>

    <node name="red_object_detector_node"
          pkg="robot"
          type="object_detector.py"
          output="screen"/>

    <node name="object_follower_node"
          pkg="robot"
          type="object_follower.py"
          output="screen"/>

</launch>
```

rqt_graph