Évasion d'obstacles et Véhicule Braitenberg

Configuration préalable

1. **Mise à jour de l'environnement ROS2** :

```
# Dans le répertoire Home/TD_ws
mkdir -p ~/TD_ws/src
cd ~/TD_ws/src
git clone https://bitbucket.org/theconstructcore/exploring-ros2-with-wheeled-robot.git
cd ..
colcon build --symlink-install --packages-select my_package

# Configurer TurtleBot3 et Gazebo
source /opt/ros/humble/setup.bash
export TURTLEBOT3_MODEL=burger
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:$(ros2 pkg prefix turtlebot3_gazebo)/share/turtlebot3_gazebo/models/
```

2.a) Test de l'exemple d'évitement d'obstacles

Structure du code existant (`obstacle_avoidance.cpp`)

Le noeud analyse les données LiDAR et contrôle les vitesses linéaire/angulaire en fonction des obstacles.

Étapes :

1. **Lancer le simulateur Gazebo** :

```
export TURTLEBOT3_MODEL=burger
ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

2. **Exécuter le noeud d'évitement** :

```
source install/setup.bash
ros2 run my_package obstacle_avoidance -ros-args -p base_speed:=0.2 -p gain:=1.0
```

Explication du code clé :

```
// obstacle_avoidance.cpp
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/laser_scan.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include <vector>
#include <cmath>
#include <algorithm>

class ObstacleAvoidance : public rclcpp::Node {
public:
    ObstacleAvoidance() : Node("obstacle_avoidance") {
        // Declare parameters
        declare_parameter("num_zones", 5);
        declare_parameter("front_threshold", 0.5);
        declare_parameter("side_threshold", 0.3);
        declare_parameter("max_linear_speed", 0.2);
        declare_parameter("turn_speed", 0.5);
        declare_parameter("filter_coefficient", 0.3);

        // Get parameters
        num_zones_ = get_parameter("num_zones").as_int();
        front_threshold_ = get_parameter("front_threshold").as_double();
        side_threshold_ = get_parameter("side_threshold").as_double();
        max_linear_speed_ = get_parameter("max_linear_speed").as_double();
        turn_speed_ = get_parameter("turn_speed").as_double();
        filter_coefficient_ = get_parameter("filter_coefficient").as_double();

        // Initialize last command
        last_cmd_.linear.x = 0.0;
        last_cmd_.angular.z = 0.0;

        // Initialize zone distances with high values
        min_dist_per_zone_.resize(num_zones_, INFINITY);

        // Subscription to LiDAR
        laser_sub_ = create_subscription<sensor_msgs::msg::LaserScan>(
            "/scan", 10, std::bind(&ObstacleAvoidance::process_scan, this,
std::placeholders::_1));

        // Publication to cmd_vel
        cmd_pub_ = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);

        // Timer for publishing commands (10Hz)
        timer_ = create_wall_timer(
            std::chrono::milliseconds(100),
            std::bind(&ObstacleAvoidance::publish_command, this));

        RCLCPP_INFO(get_logger(), "Obstacle Avoidance node initialized with %d
zones", num_zones_);
    }

private:
    void process_scan(const sensor_msgs::msg::LaserScan::SharedPtr scan) {
        if (scan->ranges.empty()) {
            RCLCPP_WARN(get_logger(), "Received empty laser scan");
            return;
        }

        // Reset min distances
        std::fill(min_dist_per_zone_.begin(), min_dist_per_zone_.end(), INFINITY);

        // Calculate samples per zone, ensure at least 1 sample per zone
        size_t samples_per_zone = std::max(size_t(1), scan->ranges.size() /
num_zones_);

        // Process scan data
        for (size_t i = 0; i < scan->ranges.size(); ++i) {
            // Calculate zone index
            int zone = std::min(static_cast<int>(i / samples_per_zone), num_zones_
- 1);

            // Get range value
            float range = scan->ranges[i];

            // Filter invalid readings
            if (std::isfinite(range) && range >= scan->range_min && range <= scan-
>range_max) {
                min_dist_per_zone_[zone] = std::min(min_dist_per_zone_[zone],
range);
            }
        }

        // Log distances for debugging (using debug log level directly)
        RCLCPP_DEBUG(get_logger(), "Zone distances: %f %f %f %f %f",
                    num_zones_ > 0 ? min_dist_per_zone_[0] : INFINITY,
                    num_zones_ > 1 ? min_dist_per_zone_[1] : INFINITY,
                    num_zones_ > 2 ? min_dist_per_zone_[2] : INFINITY,
                    num_zones_ > 3 ? min_dist_per_zone_[3] : INFINITY,
                    num_zones_ > 4 ? min_dist_per_zone_[4] : INFINITY);
    }

    void publish_command() {
        geometry_msgs::msg::Twist cmd;

        // Default behavior: move forward
        cmd.linear.x = max_linear_speed_;
        cmd.angular.z = 0.0;

        // Check if we have valid scan data
        bool valid_data = false;
        for (const auto& dist : min_dist_per_zone_) {
            if (std::isfinite(dist)) {
                valid_data = true;
                break;
            }
        }
        if (!valid_data) {
            // Use throttle_duration instead of THROTTLE macro which might not be
available
            static rclcpp::Time last_warning_time = now();
            if ((now() - last_warning_time).seconds() >= 1.0) {
                RCLCPP_WARN(get_logger(), "No valid distance data available,
stopping robot");
                last_warning_time = now();
            }
            cmd.linear.x = 0.0;
            cmd_pub_->publish(cmd);
            return;
        }

        // Get the front zones (center zones)
        int front_start = (num_zones_ - 1) / 2 - (num_zones_ % 2 == 0 ? 0 : 1);
        int front_end = (num_zones_ - 1) / 2 + 1;

        // Get the minimum distance in front
        float front_min = INFINITY;
        for (int i = front_start; i <= front_end; ++i) {
            if (i >= 0 && i < num_zones_ && min_dist_per_zone_[i] < front_min) {
                front_min = min_dist_per_zone_[i];
            }
        }

        // Determine left and right distances
        float left_min = INFINITY;
        float right_min = INFINITY;

        for (int i = 0; i < front_start; ++i) {
            left_min = std::min(left_min, min_dist_per_zone_[i]);
        }

        for (int i = front_end + 1; i < num_zones_; ++i) {
            right_min = std::min(right_min, min_dist_per_zone_[i]);
        }

        // Obstacle avoidance logic
        if (front_min < front_threshold_) {
            // Obstacle in front, need to turn
            cmd.linear.x = std::max(0.0, max_linear_speed_ * (front_min /
front_threshold_));

            // Decide which way to turn (prefer the side with more space)
            if (left_min > right_min) {
                cmd.angular.z = turn_speed_; // Turn left
                RCLCPP_DEBUG(get_logger(), "Obstacle ahead, turning left");
            } else {
                cmd.angular.z = -turn_speed_; // Turn right
                RCLCPP_DEBUG(get_logger(), "Obstacle ahead, turning right");
            }
        } else if (left_min < side_threshold_) {
            // Obstacle on the left, turn right
```

```cpp
            cmd.angular.z = -turn_speed_ * (1.0 - left_min / side_threshold_);
            RCLCPP_DEBUG(get_logger(), "Obstacle on the left, adjusting right");
        } else if (right_min < side_threshold_) {
            // Obstacle on the right, turn left
            cmd.angular.z = turn_speed_ * (1.0 - right_min / side_threshold_);
            RCLCPP_DEBUG(get_logger(), "Obstacle on the right, adjusting left");
        }

        // Apply low-pass filter for smooth transitions
        cmd.linear.x = filter_coefficient_ * cmd.linear.x + (1.0 -
filter_coefficient_) * last_cmd_.linear.x;
        cmd.angular.z = filter_coefficient_ * cmd.angular.z + (1.0 -
filter_coefficient_) * last_cmd_.angular.z;

        // Store command for next filter
        last_cmd_ = cmd;

        // Publish command
        cmd_pub_->publish(cmd);
    }

    // Subscriptions and Publishers
    rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr laser_sub_;
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_pub_;
    rclcpp::TimerBase::SharedPtr timer_;

    // Parameters
    int num_zones_;
    double front_threshold_;
    double side_threshold_;
    double max_linear_speed_;
    double turn_speed_;
    double filter_coefficient_;

    // State variables
    std::vector<float> min_dist_per_zone_;
    geometry_msgs::msg::Twist last_cmd_;
};

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    auto node = std::make_shared<ObstacleAvoidance>();
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

2.b) Transformation en Véhicule Braitenberg

Principe

Connexion directe capteurs-moteurs avec inversion gauche/droite :

- Capteurs gauches → Moteur droit
- Capteurs droits → Moteur gauche

Nouveau code (`braitenberg_avoidance.cpp`)

```
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/laser_scan.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include <vector>
#include <algorithm>
#include <cmath>

class BraitenbergVehicle : public rclcpp::Node {
public:
    BraitenbergVehicle() : Node("braitenberg_vehicle") {
        // Déclaration des paramètres configurables
        this->declare_parameter("base_speed", 0.15);
        this->declare_parameter("max_linear_speed", 0.3);
        this->declare_parameter("max_angular_speed", 1.0);
        this->declare_parameter("min_safe_distance", 0.3);
        this->declare_parameter("gain", 0.8);
        this->declare_parameter("smoothing_factor", 0.7);

        // Récupération des paramètres
        base_speed_ = this->get_parameter("base_speed").as_double();
        max_linear_speed_ = this->get_parameter("max_linear_speed").as_double();
        max_angular_speed_ = this->get_parameter("max_angular_speed").as_double();
        min_safe_distance_ = this->get_parameter("min_safe_distance").as_double();
        gain_ = this->get_parameter("gain").as_double();
        smoothing_factor_ = this->get_parameter("smoothing_factor").as_double();

        // Initialisation des vitesses
        current_linear_x_ = 0.0;
        current_angular_z_ = 0.0;

        // Abonnement au topic du scanner laser
        laser_sub_ = create_subscription<sensor_msgs::msg::LaserScan>(
            "/scan", 10, std::bind(&BraitenbergVehicle::process_scan, this,
std::placeholders::_1));

        // Publication des commandes de vitesse
        cmd_pub_ = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);

        // Timer pour publier les commandes à fréquence fixe
        timer_ = this->create_wall_timer(
            std::chrono::milliseconds(50),  // 20Hz
            std::bind(&BraitenbergVehicle::publish_command, this));

        RCLCPP_INFO(this->get_logger(), "Braitenberg vehicle node initialized");
        RCLCPP_INFO(this->get_logger(), "Base speed: %.2f, Gain: %.2f",
base_speed_, gain_);
    }

private:
    void process_scan(const sensor_msgs::msg::LaserScan::SharedPtr scan) {
        if (scan->ranges.empty()) {
            RCLCPP_WARN(this->get_logger(), "Received empty laser scan");
            return;
        }

        // Division en deux zones (gauche/droite)
        const size_t mid_index = scan->ranges.size() / 2;

        // Filtrage et calcul des distances minimales
        float left_min = calculate_filtered_min(scan, 0, mid_index);
        float right_min = calculate_filtered_min(scan, mid_index, scan-
>ranges.size());

        // Détection d'obstacle imminent
        bool emergency = false;
        for (size_t i = 0; i < scan->ranges.size(); ++i) {
            // Vérifier uniquement les angles frontaux (approximativement ±45°)
            if ((i < scan->ranges.size() / 8) || (i > 7 * scan->ranges.size() /
8)) {
                if (is_valid_range(scan->ranges[i], scan->range_min, scan-
>range_max) &&
                    scan->ranges[i] < min_safe_distance_) {
                    emergency = true;
                    RCLCPP_WARN_THROTTLE(this->get_logger(),
                                         *this->get_clock(),
                                         1000, // Afficher max 1 fois par seconde
                                         "Emergency: Obstacle too close!");
                    break;
                }
            }
        }

        // Mise à jour des commandes de vitesse
        geometry_msgs::msg::Twist cmd;

        if (emergency) {
            // Comportement d'urgence : arrêt et rotation
            cmd.linear.x = -0.05;  // Léger recul

            // Déterminer la direction de rotation en fonction de l'espace
disponible
            if (left_min > right_min) {
                cmd.angular.z = 0.8;  // Rotation à gauche
            } else {
                cmd.angular.z = -0.8; // Rotation à droite
            }
        } else {
            // Comportement normal de Braitenberg (Type 2b : Fear - connexions
croisées)
```

```
            // La vitesse de chaque roue est inversement proportionnelle à la
distance du côté opposé

            // Transformation non linéaire pour une meilleure réponse
            float left_activation = activation_function(left_min);
            float right_activation = activation_function(right_min);

            // Calculer les vitesses des roues
            float left_wheel = base_speed_ * (1.0 - gain_ * right_activation);
            float right_wheel = base_speed_ * (1.0 - gain_ * left_activation);

            // Convertir en vitesses linéaire et angulaire
            float target_linear_x = (left_wheel + right_wheel) / 2.0;
            float target_angular_z = (left_wheel - right_wheel) * 2.0;

            // Limiter les vitesses
            target_linear_x = std::max(0.0f, std::min(static_cast<float>
(max_linear_speed_), target_linear_x));
            target_angular_z = std::max(-static_cast<float>(max_angular_speed_),
                                       std::min(static_cast<float>
(max_angular_speed_), target_angular_z));

            // Application d'un lissage exponentiel pour éviter les mouvements
brusques
            current_linear_x_ = smoothing_factor_ * current_linear_x_ +
                              (1.0 - smoothing_factor_) * target_linear_x;
            current_angular_z_ = smoothing_factor_ * current_angular_z_ +
                               (1.0 - smoothing_factor_) * target_angular_z;

            cmd.linear.x = current_linear_x_;
            cmd.angular.z = current_angular_z_;
        }

        latest_cmd_ = cmd;

        RCLCPP_DEBUG(this->get_logger(),
                    "Distances - Left: %.2f, Right: %.2f | Cmd - Linear: %.2f,
Angular: %.2f",
                    left_min, right_min, cmd.linear.x, cmd.angular.z);
    }

    float calculate_filtered_min(const sensor_msgs::msg::LaserScan::SharedPtr
scan,
                               size_t start_idx, size_t end_idx) {
        std::vector<float> valid_ranges;
        valid_ranges.reserve(end_idx - start_idx);

        for (size_t i = start_idx; i < end_idx; ++i) {
            if (is_valid_range(scan->ranges[i], scan->range_min, scan->range_max))
{
                valid_ranges.push_back(scan->ranges[i]);
            }
        }
        if (valid_ranges.empty()) {
            // Si pas de mesures valides, retourner une valeur sûre
            return scan->range_max;
        }

        // Trier et calculer la moyenne des 20% plus petites valeurs
        // pour une mesure plus robuste que simplement prendre le minimum
        std::sort(valid_ranges.begin(), valid_ranges.end());
        size_t num_samples = std::max(size_t(1), valid_ranges.size() / 5);

        float sum = 0.0f;
        for (size_t i = 0; i < num_samples; ++i) {
            sum += valid_ranges[i];
        }

        return sum / num_samples;
    }

    bool is_valid_range(float range, float min_range, float max_range) {
        return std::isfinite(range) && range >= min_range && range <= max_range;
    }

    float activation_function(float distance) {
        // Fonction d'activation non-linéaire (sigmoid inversée)
        // - Forte réponse pour distances faibles
        // - Réponse diminuant rapidement avec la distance
        float scale = 2.0f;
        float midpoint = 1.0f;

        // Pour éviter les divisions par zéro
        distance = std::max(0.1f, distance);

        return 1.0f / (1.0f + std::exp(scale * (distance - midpoint)));
    }

    void publish_command() {
        // Publier périodiquement la dernière commande calculée
        cmd_pub_->publish(latest_cmd_);
    }

    // Abonnements et publications
    rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr laser_sub_;
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_pub_;
    rclcpp::TimerBase::SharedPtr timer_;

    // Paramètres
    double base_speed_;
    double max_linear_speed_;
    double max_angular_speed_;
    double min_safe_distance_;
    double gain_;
    double smoothing_factor_;

    // État
    double current_linear_x_;
    double current_angular_z_;
    geometry_msgs::msg::Twist latest_cmd_;
};

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<BraitenbergVehicle>());
    rclcpp::shutdown();
    return 0;
}
```

Modification du `CMakeLists.txt`

```
# Ajouter après les autres exécutables
add_executable(braitenberg_avoidance src/braitenberg_avoidance.cpp)
ament_target_dependencies(braitenberg_avoidance rclcpp sensor_msgs geometry_msgs)

install(TARGETS
    obstacle_avoidance
    braitenberg_avoidance
    DESTINATION lib/${PROJECT_NAME}
)
```

Compilation et exécution

1. **Compiler le package** :

```
cd ~/TD_ws
colcon build --packages-select my_package
source install/setup.bash
```

2. **Lancer le Braitenberg** :

```
source install/setup.bashv
ros2 run my_package braitenberg_avoidance -ros-args -p base_speed:=0.2 -p
gain:=1.0
```

Résultats

- **Obstacle Avoidance (2a)** : Le robot s'arrête et tourne lorsqu'un obstacle est détecté devant.

- **Braitenberg (2b)** : Le robot ajuste continuellement sa trajectoire sans s'arrêter, avec un comportement plus fluide.

Comparaison des deux approches

Critère	Obstacle Avoidance (2a)	Braitenberg Vehicle (2b)
Type de contrôle	Logique conditionnelle (if/else)	Connexion directe capteurs-moteurs
Réaction aux obstacles	Arrêt + rotation brutale	Ajustement proportionnel continu
Paramètres clés	`front_threshold`, `turn_speed`	`gain`, `base_speed`
Dépendance temporelle	Temps de réaction fixe (10Hz)	Réactivité adaptative (20Hz + lissage)
Mouvement	Saccadé (arrêts/redémarrages)	Fluide (pas d'arrêt complet)
Complexité code	150 lignes (+ gestion explicite des zones)	120 lignes (+ calculs matriciels implicites)
Avantages	Comportement prévisible	Évite les oscillations
Inconvénients	Risque de blocage devant obstacles complexes	Moins bon en espaces très exigus
Usage des capteurs	3-5 zones prédéfinies	Tout le champ LiDAR (360°)
Type de comportement	Réflexe	Émergent

> **Note** : Ajustez le `gain` et les zones LiDAR dans le code pour optimiser le comportement.
