Spiegazione dettagliata del codice — *Lidar Pose Estimation Node*

Scopo: questa spiegazione passo-passo descrive il comportamento del nodo ROS2 `LidarListener` per l'elaborazione di dati LIDAR. Il testo è pensato per essere inserito in un paper; contiene una descrizione funzionale dei metodi, annotazioni su scelte progettuali, limiti, possibili miglioramenti e suggerimenti sperimentali. Puoi accorciare o riformulare i paragrafi secondo le esigenze della pubblicazione.

Sommario esecutivo

Il nodo `LidarListener` si occupa di:

- sottoscrivere i topic `LaserScan` e `PointCloud2` di un sensore LIDAR;
- applicare un filtro voxel (VoxelGrid) per ridurre la densità del point cloud;
- filtrare i punti in base all'angolo orizzontale;
- convertire la rappresentazione cartesiana in coordinate polari (opzionale/locale);
- eseguire una versione custom di RANSAC per segmentare piani (2D e 3D, implementazioni proprie);
- separare i punti in `plane` (piano stimato) e `obstacles` (resto);
- applicare clustering euclideo (implementazione custom e due versioni basate su PCL) per trovare oggetti/cluster;
- pubblicare i point cloud segmentati sui topic `/reacsa/plane_points` e `/reacsa/obstacle_points`.

Questa pipeline è pensata per la stima rapida della presenza di piani e ostacoli a partire da scansioni LIDAR in tempo (quasi) reale.

Struttura generale del nodo

- **Costruttore** `LidarListener()`: crea le sottoscrizioni e i publisher ROS2:
- `laser_sub_` su `/reacsa/scan` (LaserScan) con QoS `best_effort`;
- `pc_sub_` su `/reacsa/velodyne_points` (PointCloud2) con QoS `best_effort`;
- publisher `plane_pub_` e `obstacle_pub_` per pubblicare i risultati come `PointCloud2`.
- **Callback** `laserScanCallback`: semplicemente logga alcuni parametri e le prime range per debug.
- **Callback** `pointCloudCallback`: è il cuore della pipeline — riceve il `PointCloud2`, lo converte in `pcl::PointCloud<pcl::PointXYZ>`, applica filtraggio voxel, segmentazione RANSAC per piani, clustering degli ostacoli e pubblica i risultati.

Funzioni principali (analizzate passo-passo)

`filterPointCloud( cloud, filterRes )`

Tipo template: `template<typename PointT>`

Scopo: applica un filtro VoxelGrid e filtra i punti per angolo orizzontale, restituendo un point cloud ridotto e limitato ad una finestra angolare.

Passi logici: 1. registra `startime` per misurare il tempo di esecuzione; 2. crea un oggetto `pcl::VoxelGrid<PointT> vg` e un nuovo `cloud_filtered`; 3. chiama `vg.setInputCloud(cloud)` e `vg.setLeafSize(filterRes, filterRes, filterRes)` seguita da `vg.filter(*cloud_filtered)` — questo sostituisce tutti i punti contenuti in ogni voxel con il loro centroide (riduzione di densità); 4. itera su `cloud_filtered->points` e calcola l'angolo orizzontale `angle = atan2(y, x)`; 5. mantiene nel risultato (`cloud_region`) solo i punti con `|angle| <= 3 * M_PI / 4` (nota: vedi *incongruenza commento/condizione* sotto); 6. imposta `width`, `height`, `is_dense` e misura il tempo totale.

Osservazioni e punti importanti per paper: - Il filtro VoxelGrid è una pratica standard per ridurre la quantità di dati senza alterare significativamente la forma globale delle superfici. `filterRes` è il parametro primario da tarare (qui es. 0.01 m). - La soglia angolare limita i punti a una finestra frontale: la condizione utilizzata è `|angle| <= 3π/4` (±135°). Tuttavia nel codice ci sono commenti contraddittori ("[-3/8 pi] -> [-67.5,67.5]"), quindi **verificare** quale comportamento si desidera e correggere il commento o la condizione. - Costi computazionali: il filtro voxel ha costo O(N) per punto per la costruzione della mappa dei voxel (in pratica efficiente). L'iterazione angolare è O(N_filtered). - Suggerimenti sperimentali: confrontare `filterRes` tra 0.01 e 0.05 m per bilanciare accuratezza e latenza.

`toPolar( cloud )`

Tipo template: `template<typename PointT>`

Scopo: converte un point cloud cartesiano in una rappresentazione polare semplice (`PolarPoint` con r, theta, phi e indice).

Dettagli implementativi: - `r = sqrt(x^2 + y^2 + z^2)` (distanza radiale); - `theta = atan2(y, x)` (angolo azimutale nel piano xy); - `phi = atan2(z, sqrt(x^2 + y^2))` (angolo elevazione); - `idx = &point - &cloud->points[0]` calcola l'indice dalla posizione in memoria.

Note e potenziali problemi: - L'uso di `&point - &cloud->points[0]` sfrutta che `point` è una reference all'elemento nel vettore sottostante; rimane funzionante ma meno leggibile: preferire un ciclo indicizzato per chiarezza e sicurezza (o `std::distance`). Se `cloud->points` è vuoto, l'espressione è indefinita; il codice assume cloud non vuoto. - L'output è un vettore di `PolarPoint` utile per analisi angolari o metodi che beneficiano di ordinamenti angolari.

`Ransac2D( cloud, maxIterations, distanceTol )`

Tipo template: `template<typename PointT>`

Scopo: implementazione custom di RANSAC per adattare una linea 2D ai punti proiettati sul piano xy.

Algoritmo: 1. Per ogni iterazione: seleziona casualmente due punti distinti (ind1, ind2); 2. Costruisce l'equazione della retta passante per p1 e p2; 3. Calcola la distanza di ogni punto dalla retta e se inferiore a `distanceTol` lo segna come inlier; 4. Tiene il miglior insieme di inlier su tutte le iterazioni.

Osservazioni: - L'implementazione usa `pcl::PointXYZ` per estrarre coordinate anche se la funzione è template — questo limita la generalità quando PointT ≠ pcl::PointXYZ. - L'uso di `srand(time(NULL))` dentro la funzione rende non riproducibile l'esperimento a meno di salvare o controllare il seme esternamente. Per riproducibilità in pubblicazioni preferire `std::mt19937` con seme controllato. - Complessità: O(k * N) dove k = maxIterations e N = numero di punti.

`Ransac3d( cloud, maxIterations, distanceTol )`

Tipo template: `template<typename PointT>`

Scopo: implementazione custom di RANSAC per l'adattamento di un piano in 3D.

Algoritmo: 1. Per `i` in `0..maxIterations-1`: - sceglie tre indici distinti (ind1, ind2, ind3); - calcola il vettore normale del piano usando il prodotto vettoriale tra (p2-p1) e (p3-p1): questo produce (A, B, C) e poi D per l'equazione `Ax + By + Cz + D = 0`; - se il denominatore (norma del vettore normale) è zero (punti collineari) salta l'iterazione; - per ogni punto calcola `dist = |A x + B y + C z + D| / ||(A,B,C)||` e lo aggiunge agli inlier se `dist < distanceTol`; - aggiorna `inliersResult` se la cardinalità di `inliers` è la migliore trovata.

Osservazioni critiche: - Come per la versione 2D, `srand(time(NULL))` è usato; usare motori RNG moderni per riproducibilità e qualità statistica. - L'approccio è corretto e chiaro; tuttavia, PCL fornisce `pcl::SACSegmentation` con modelli robusti e ottimizzati per questa operazione (usare PCL-built-ins può essere più efficiente e offre metodi aggiuntivi come RANSAC con refinamento LMedS, ecc.). - Un compromesso da valutare: la tua implementazione è utile quando si desidera piena comprensione e controllo sul procedimento (ad esempio per pubblicare un metodo didattico o aggiungere vincoli personalizzati), mentre per prestazioni e qualità in produzione è preferibile PCL.

`SegmentPlane( cloud, maxIterations, distanceThreshold )`

Tipo template: `template<typename PointT>`

Scopo: separa un point cloud nelle due classi `plane` e `obstacles` usando il RANSAC 3D.

Passi: 1. Converte il cloud cartesiano in coordinate polari usando `toPolar`; costruisce così `polarCloud` dove `x=r`, `y=theta`, `z=phi`; 2. Esegue `Ransac3d` sul `polarCloud` per ottenere gli indici degli inlier; 3. Usa gli indici ottenuti per separare i punti originali (cartesiani) in `planeCloud` (inliers) e `obsCloud` (outliers); 4. Logga informazioni e tempi.

Punto critico da documentare nel paper: - **Trasformazione in polar coordinates prima del RANSAC 3D** è una scelta progettuale significativa: si effettua la stima di un "piano" nello spazio (r, theta, phi) invece che nello spazio (x, y, z). Questa scelta va motivata (ad es. per rendere lineari certe relazioni o per stabilizzare la stima del piano rispetto a fenomeni radiali). Se la motivazione non è chiara, è probabile che la segmentazione in polari mischi grandezze con unità diverse (metri vs radianti) e renda la soglia `distanceThreshold` difficile da interpretare. - Se lo scopo è segmentare piani geometrici reali (terra, muri), tipicamente si applica RANSAC direttamente su (x, y, z) (o su una proiezione) perché i piani reali sono iperplanari nello spazio cartesiano. Motiva chiaramente la scelta polare o valuta un confronto sperimentale (cartesiano vs polare) e riportalo nei risultati.

Clustering — `proximity`, `proximity_iterative`, `euclideanCluster`, `euclideanClusterOptimized`

Scopo: trovare gruppi/cluster di punti che rappresentino singoli ostacoli.

Implementazioni: - `proximity` è la versione ricorsiva classica che marca i punti processati e espande il cluster chiamando ricorsivamente la funzione per i vicini. - `proximity_iterative` è la versione iterativa (stack-based) che evita profondità di ricorsione e usa un buffer riutilizzabile `nearby_buffer` per ridurre allocazioni ripetute; è più scalabile per grandi point cloud. - `euclideanCluster` e `euclideanClusterOptimized` chiamano rispettivamente la versione ricorsiva e quella iterativa per costruire i cluster con una `KdTree` custom (metodo `search` per trovare i vicini entro `distanceTol`).

Ottimizzazioni: - `euclideanClusterOptimized` usa `std::vector<char> processed` anziché cercare con `std::find` nell'array `processed` come nella versione semplice: questo riduce il costo da O(N) per controllo a O(1). - L'uso di un `nearby_buffer` riutilizzabile riduce allocazioni dinamiche durante il clustering.

Note implementative: - Il KdTree è custom (presumibilmente una semplice implementazione ad albero binario). Per grandi dataset usare `pcl::search::KdTree` migliora costruttività e performance. - I parametri `distanceTol`, `minSize`, `maxSize` sono critici per discriminare rumore e oggetti veri; nel codice es. `clusterTolerance = 0.5f` (50 cm) è una scelta abbastanza ampia — va rapportata alla densità del cloud e al voxel size.

`Clustering( obsCloud, clusterTolerance, minSize, maxSize )` (custom)

Flusso: 1. Converte `obsCloud` in `std::vector<std::vector<float>> points` per la KD-tree custom; 2. Crea la KD-tree, inserisce i punti e chiama `euclideanCluster` per ottenere insiemi di indici; 3. Per ciascun cluster valido costruisce `pcl::PointCloud<PointT>::Ptr` e lo aggiunge ai risultati.

Osservazioni: - Implementazione chiara, ma la costruzione della KD-tree punto per punto può essere lenta; una build bulk è preferibile quando disponibile. - La funzione ritorna i cluster filtrando per dimensione.

`ClusteringPCL( obsCloud, clusterTolerance, minSize, maxSize )`

Scopo: versione che usa gli strumenti di PCL (`pcl::search::KdTree` e `pcl::EuclideanClusterExtraction`) che sono ottimizzati e manutenuti.

Consiglio per il paper: presentare risultati comparativi (tempo e qualità) tra la versione custom e quella PCL per motivare la scelta finale.

`ClusteringOptimized` **(versione migliorata)**

Migliorie: - usa `std::array<float,3>` come formato compatto; - mantiene `tree_` come membro di classe per riutilizzo (se supportato) per evitare ricostruzioni ad ogni callback; - usa la versione iterativa del clustering (`euclideanClusterOptimized`) e buffer preallocati per prestazioni migliori.

Nota: nel codice la logica di ricostruzione della `tree_` cancella e ricrea l'oggetto (la sezione commentata suggerisce che sarebbe preferibile avere un metodo `clear()` interno alla KD-tree per riusare la struttura): implementare `clear()` nel KdTree è raccomandato.

`pointCloudCallback` **— pipeline end-to-end**

Passi principali (come implementati nel callback): 1. log iniziale dei metadati del `PointCloud2` ricevuto; 2. mostra i primi 5 punti con iteratori `PointCloud2ConstIterator<float>` (per debug); 3. conversione ROS→PCL (`pcl::fromROSMsg`); 4. filtro voxel tramite `filterPointCloud` (v. sopra) con `voxelSize = 0.01f` (1 cm) — parametro di test; 5. segmentazione plane/outliers: `SegmentPlane(..., maxIterations = 500, distanceThreshold = 0.02f)`; 6. pubblicazione dei punti di piano in `/reacsa/plane_points`; 7. se `obsCloud` ha punti: clustering ottimizzato con `clusterTolerance = 0.5`, `minSize = 10`, `maxSize = 5000`; 8. pubblicazione del cluster più grande (se esiste) come `/reacsa/obstacle_points`. Se non ci sono cluster pubblica un `PointCloud2` vuoto (costruzione manuale del messaggio).

Nota: le scelte dei valori numerici (voxel 1cm, RANSAC 0.02 m, cluster tol 0.5 m) devono essere giustificate sperimentalmente: includi nel paper una tabella di tuning e i risultati correlati (precision/recall, tempo di esecuzione medio, frame rate).

Main

Il `main` inizializza ROS2 (`rclcpp::init`), crea e mette in esecuzione il nodo `LidarListener` con `rclcpp::spin` e poi chiude ROS con `rclcpp::shutdown`.

Criticità, bug e note di manutenzione (da includere nel paper)

1. **Incoerenza commento/condizione angoli**: il codice commenta due valori diversi rispetto alla condizione effettiva. Verificare se l'intervallo desiderato è ±67.5° o ±135° e correggere commento o condizione.

2. **Uso di** `srand(time(NULL))` dentro le funzioni RANSAC rende i risultati non riproducibili e può essere chiamato più volte — usare una singola istanza RNG con `std::mt19937` e seme controllabile esternamente.
3. **Miscelazione di unità in SegmentPlane**: l'uso di coordinate polari (`r` in metri, `theta` / `phi` in radianti) per RANSAC 3D implica una metrica non omogenea. Questo rende `distanceThreshold` meno interpretabile. Motivare la scelta o usare normalizzazione/scaling.
4. **Template vs tipi concreti**: alcune funzioni template (es. Ransac2D) usano internamente `pcl::PointXYZ` — chiarire o generalizzare l'implementazione se si vuole mantenere la generalità.
5. **Sicurezza sui puntatori/indici**: `&point - &cloud->points[0]` assume cloud non vuoto; aggiungere check `if (cloud->points.empty()) return {}`.
6. **Gestione della KD-tree**: attualmente la ricostruzione viene fatta ogni chiamata; mantenere una struttura con `clear()` e `bulk insert` migliora latenza.
7. **Thread-safety**: se il nodo viene usato in un executor multithread, attenzione all'accesso concorrente a `tree_` e alle variabili membro; proteggere con mutex o usare struttura per thread-local.

Suggerimenti per la sezione "Metodi" del paper

- **Descrivere la pipeline**: illustrare la sequenza di passi (voxel filter → angolo filter → RANSAC plane → clustering) e giustificare ciascuno.
- **Parametri principali**: inserire una tabella con `voxelSize`, `maxIterations`, `distanceThreshold`, `clusterTolerance`, `minSize`, `maxSize` e valori di riferimento usati negli esperimenti. Esempio: `voxelSize=0.01 m`, `maxIterations=500`, `distanceThreshold=0.02 m`, `clusterTolerance=0.5 m`.
- **Valutazione sperimentale**: riportare grafici/metriche su:
- tempo medio per callback (ms) prima/dopo ottimizzazione;
- numero di punti prima/dopo filtraggio;
- accuratezza della segmentazione del piano (es. percentuale di inlier corretti su dataset con ground truth);
- confronto clustering custom vs PCL per tempo e risultato.
- **Aggiungere figure**: esempio di pipeline, esempio di point cloud prima/dopo segmentazione (immagini colore: piano vs ostacoli), diagramma del KdTree e dei clusters rilevati.

Miglioramenti consigliati (lista attuabile)

1. **Sostituire RNG**: usare `std::mt19937` inizializzato una volta (es. membro della classe) per riproducibilità.
2. **RANSAC con PCL per confronto**: valutare `pcl::SACSegmentation<pcl::PointXYZ>` e `pcl::ExtractIndices` come baseline.
3. **Normalizzare coordinate polari o eseguire RANSAC in cartesiano**: evitare miscele di unità. Se si mantiene spazio polare, scalare `theta` / `phi` in modo che le distanze siano confrontabili.
4. **Bulk KD-tree build / clear**: implementare `clear()` e bulk insert nella KD-tree custom oppure usare la `pcl::search::KdTree` per performance migliori.
5. **Misure di qualità**: integrare test automatici su dataset di riferimento e misurare latenza e accuratezza.
6. **Controlli addizionali**: gestire casi limite (cloud vuoto), evitare dereferenziazioni non sicure.

Esempio di paragrafo tecnico (da inserire direttamente nel paper)

> Il nodo `LidarListener` processa il `PointCloud2` proveniente dal sensore LIDAR tramite una pipeline composta da (i) VoxelGrid filtering per ridurre la densità dei punti, (ii) filtro angolare per limitare l'analisi all'area di interesse frontale, (iii) segmentazione di piano mediante un algoritmo RANSAC implementato ad hoc e (iv) clustering euclideo per isolare oggetti discreti. I parametri principali sono il `voxelSize` del filtro (tipicamente 0.01 m), il numero di iterazioni di RANSAC (tipicamente 500) e la soglia di distanza per gli inlier (tipicamente 0.02 m). Per la fase di clustering sono stati testati sia un algoritmo custom basato su KD-tree che la funzione `pcl::EuclideanClusterExtraction`; nei risultati confrontiamo latenza e qualità dei due approcci.

Tabella suggerita dei parametri (da includere nel paper)

Parametro	Valore di riferimento	Note
voxelSize	0.01 m	compromesso accuratezza/velocità
maxIterations (RANSAC)	500	qualità dell'adattamento del piano
distanceThreshold (RANSAC)	0.02 m	soglia in metri per inlier (se su cartesiano)
clusterTolerance	0.5 m	distanza massima tra punti nello stesso cluster
minSize	10 punti	evita rumore
maxSize	5000 punti	evita cluster troppo grandi

Conclusione

Ho descritto dettagliatamente ogni funzione, le scelte progettuali, i limiti osservati e i possibili miglioramenti. Per il paper suggerisco di aggiungere una sezione sperimentale che giustifichi le scelte dei parametri con risultati quantitativi (tempo, accuratezza di segmentazione, confronto con metodi PCL). Se desideri, posso generare una versione ridotta (sintetica) di questo testo adatta alla lunghezza e al tono di una conference o di un journal.

Fine del file.