ALGORITMO ODOMÉTRICO LOAM INNOVADO DESDE C++ SIN ROBOT OPERATIVE SYSTEM

Víctor Joel Pinargote Bravo[1]
vpinargote@espam.edu.ec
Alfonso Tomás Loor Vera[1]
aloor@espam.edu.ec
Edwin Wellington Moreira Santos[1]
edwinmoreira97@espam.edu.ec
Luisa Anabel Palacios López[2]
luisa.palacios@unesum.edu.ec

[1]Escuela Superior Politécnica Agropecuaria de Manabí Manuel Félix López, ESPAM MFL. Carrera de Computación. Calceta, Ecuador.

[2]Universidad Estatal del Sur de Manabí, UNESUM. Carrera de Ingeniería Ambiental. Jipijapa, Ecuador.

DOI: https://doi.org/10.56124/encriptar.v7i14.006

Resumen

La odometría desempeña un papel crucial en la estimación de la posición relativa de un robot móvil, lo que motiva este artículo a implementar el método utilizado en el algoritmo LOAM, desarrollado dentro del marco de ROS (Robot Operating System). El enfoque de este trabajo se centra en el análisis del algoritmo LOAM y su posterior implementación en C++ sin la dependencia de ROS, empleando bibliotecas como PCL (Point Cloud Library), Eigen 3.0, y algunas funcionalidades adaptadas para su uso en C++, como ROS::tf. Para llevar a cabo el análisis del algoritmo LOAM, se aplicó un proceso de ingeniería inversa y se revisó la literatura existente, basándose en los artículos publicados por los autores del algoritmo mencionado anteriormente. Asimismo, se examinaron los métodos de Visual SLAM (Simultaneous Localization and Mapping) con el fin de compararlos y proponer posibles mejoras y observaciones en el algoritmo estudiado y posteriormente implementado. Se demostró que el algoritmo LOAM presenta una mayor eficiencia en tramos rectos en comparación con las curvas, debido a la ausencia de un sensor de medida inercial que sea capaz de compensar la rotación del láser en las curvas. Por lo tanto, se recomienda realizar un análisis segmentado en escenarios con diversos tipos de recorridos para mejorar los resultados y reducir el error total del experimento.

Palabras clave: odometría, algoritmo LOAM, robot móvil.

LOAM ODOMETRIC ALGORITHM INNOVATED FROM C++ WITHOUT ROBOT OPERATIVE SYSTEM

ABSTRACT

Odometry plays a crucial role in estimating the relative position of a mobile robot, which motivates this article to implement the method used in the LOAM algorithm, developed within the ROS (Robot Operating System) framework. The focus of this work is on the analysis of the LOAM algorithm and its subsequent implementation in C++ without dependency on ROS, using libraries such as PCL (Point Cloud Library), Eigen 3.0, and some functionalities adapted for use in C++, such as ROS. ::tf. To carry out the analysis of the LOAM algorithm, a reverse engineering process was applied and the existing literature was reviewed, based on the articles published by the authors of the aforementioned algorithm. Likewise, Visual SLAM (Simultaneous Localization and Mapping) methods were examined in order to compare them and propose possible improvements and observations in the algorithm studied and subsequently implemented. It was shown that the LOAM algorithm has greater efficiency in straight sections compared to curves, due to the absence of an inertial measurement sensor that is capable of compensating for the rotation of the laser in curves. Therefore, it is recommended to perform a segmented analysis in scenarios with various types of paths to improve the results and reduce the total error of the experiment.

Keywords: odometry, LOAM algorithm, mobile robot.

1. Introducción

En la actualidad el área de la robótica y la conducción autónoma convergen, debido a que algunos de los algoritmos más eficientes de conducción independiente se implementan en ROS; siendo este el punto de partida para múltiples proyectos e investigaciones, lo que representa una problemática para ciertos grupos de desarrolladores e investigadores, debido a que muchos tienen la necesidad de empezar a investigar en esa área de la tecnología sin usar el denominado Robot Operative System (Pinargote, 2017).

La solución que aquí se presenta proporciona un cálculo de Odometría Lidar en tiempo real con el uso de elementos computacionales de última generación, ese algoritmo de cálculo puede ser utilizado como un sensor independiente sin la necesidad de añadir otro, aunque también integra funciones para usar un IMU (Inertial Measurement Unit); por lo expuesto se enuncia como objetivo del presente artículo el de argumentar la implementación del algoritmo de Odometría del LOAM en C++ sin usar ROS para obtener un array que almacene la ruta desplazada por el sensor (Zhang y Singh, 2014).

Este artículo proviene de una investigación previa titulada “Implementación del algoritmo de odometría LOAM con C++ en Linux” el cual se enmarca en el ámbito de la robótica móvil y la conducción autónoma, dado que surge de la necesidad de conseguir una localización sin depender de los Sistemas de Posicionamiento Global (GPS). Esto se debe a que existen entornos, como túneles y otros espacios subterráneos, donde los GPS no tienen cobertura o su señal es deficiente (Pinargote, 2017).

No obstante, si bien hoy existen múltiples algoritmos que tratan la problemática del SLAM (Simultaneous Localization and Mapping), tales como

los de Odometría Visual o Lidar, entre otros, la visual es el proceso mediante el que se hace una estimación del movimiento realizado por un robot móvil o por un vehículo a partir de los datos obtenidos por el sistema de visión del mismo, mientras que la Odometría Lidar, aunque posee el mismo objetivo, parte de los datos capturados por un sistema de scanner láser 3D o 2D de manera indistinta.

Hoy, la mayoría de los algoritmos de odometría visual propuestos no funcionan en tiempo real, sino que sus resultados se obtienen principalmente de manera off-line, ya que las imágenes se capturan y luego se procesan. Otros algoritmos operan a una frecuencia de captura de imágenes muy baja, lo que exige un movimiento muy lento del robot. Esto se debe a que los algoritmos de odometría visual actuales requieren cálculos muy complejos y una alta carga computacional, lo cual implica que, para ejecutarse correctamente, a menudo se necesitan equipos informáticos y sistemas de visión avanzados, lo que incrementa significativamente los costos. Por esta razón, se optó por utilizar el algoritmo V-LOAM, ya que es uno de los más confiables en este campo según el ranking de KITTI Vision Benchmark Suite.

2. Metodología (Materiales y métodos)

La metodología seguida durante el desarrollo de la investigación de este artículo es una estructura planeada previamente a la realización del trabajo y se basó en los principios científicos necesarios para un trabajo de este tipo. Primero, se realizó una revisión en profundidad de la literatura actualizada sobre la temática, con un foco de atención en el estudio de los diversos trabajos de localización basados en el cálculo de la Odometría Visual y Odometría Lidar; de lo que fueron estudiadas las ventajas e inconvenientes de cada una de las propuestas que, hasta el momento se han realizado y tienen su fundamento en lo anteriormente abordado.

Posteriormente, se optó por un método de odometría Lidar que se consideró apropiado debido a su robustez conceptual y procedimental. Se llevó a cabo un estudio detallado de cada uno de los componentes y etapas del algoritmo. Una vez que se tuvo un algoritmo de cálculo de odometría Lidar disponible, se procedió a su implementación en C++, haciendo uso de bibliotecas como PCL, Eigen3 y otras como ROSS::tf. Cabe destacar que esta última se adaptó para su uso sin ROS, lo cual se explicará más adelante.

2.1. Herramientas de software utilizadas

En primer lugar, a lo largo de este proyecto se consideró útil trabajar con la biblioteca de gran escala Point Cloud Library (PCL), la cual se distribuye bajo una licencia de Berkeley Software Distribution (BSD) y es un software de código abierto multi-plataforma, que funciona con Linux, Android, Windows, iOs y MacOS.

Para simplificar, el desarrollo PCL está compartimentado en una serie de bibliotecas de menor tamaño que pueden ser compiladas cada una por su parte. Esta modularidad es importante para su distribución en plataformas con restricciones de tamaño o cómputo (Tombari, 2013). De ella, los módulos más importantes a tomar en cuenta fueron:

• Filters: Limpia las nubes de puntos 3D al eliminar el ruido y los puntos aislados para mejorar la calidad de los datos.

• Features: Analiza y calcula las características tridimensionales de una nube de puntos para identificar patrones o elementos de interés en el entorno.

• Keypoints: Identifica y marca los puntos de interés significativos en una

imagen o nube de puntos que pueden ser útiles para el análisis o la toma de decisiones.

• Registration: Fusiona múltiples conjuntos de datos en un único modelo global para obtener una visión más completa del entorno tridimensional.

• Kd-tree: Utiliza una estructura de árbol kd para encontrar de manera eficiente los puntos más cercanos en una nube de puntos, facilitando operaciones de búsqueda.

• Octree: Genera un árbol jerárquico a partir de una nube de puntos, permitiendo la organización espacial, la reducción de la resolución y la realización de operaciones de búsqueda de manera eficiente.

• Segmentation: Agrupa los puntos de una nube en clusters para identificar objetos individuales o regiones de interés.

• Sample_consensus: Utiliza algoritmos como RANSAC y modelos geométricos para estimar relaciones espaciales entre puntos y ajustar modelos como planos y cilindros.

• Surface: Reconstruye las superficies originales a partir de los datos de escaneo 3D para obtener una representación detallada del entorno.

• Range_image: Analiza y opera con imágenes de profundidad para representar la distancia entre objetos en un entorno tridimensional.

• Io: Lee y guarda nubes de puntos desde archivos en formato PCD para facilitar el almacenamiento y el intercambio de datos 3D.

• Display: Muestra de manera visual los resultados obtenidos a partir de las nubes de datos 3D para facilitar la interpretación y el análisis de los datos (PCL, 2023).

En segundo lugar, se utilizó Eigen, una biblioteca de C++ de alto nivel basada en plantillas para álgebra lineal, operaciones con matrices y vectores, transformaciones geométricas, solucionadores numéricos y algoritmos

relacionados. Esta biblioteca es de código abierto y está licenciada bajo MPL2 desde la versión 3.1.1; se implementa utilizando la técnica de meta programación de plantillas de expresión, lo que significa que genera árboles de expresión en tiempo de compilación y concibe un código personalizado para evaluarlos. Con el uso de plantillas de expresión y un modelo de coste de operaciones en coma flotante, la biblioteca realiza su propio desenrollado y vectorización de bucle (Eigen, 2016).

Finalmente, se empleó el paquete Tf, que posibilita al usuario rastrear múltiples marcos de coordenadas a lo largo del tiempo. Este paquete mantiene la relación entre los marcos de coordenadas en una estructura de árbol temporal y permite al usuario transformar puntos, vectores y otros elementos entre dos marcos de coordenadas en cualquier momento deseado. Aunque este paquete forma parte de ROS, puede ser adaptado e implementado sin necesidad de utilizar ROS, dado que está escrito en C++ y sus dependencias son fácilmente accesibles (ROS, 2021).

2.2. Herramientas hardware utilizadas

Sobre este particular, el nuevo sensor PUCK ™ (VLP-16) de Velodyne es el producto más pequeño, nuevo y avanzado de la gama de productos 3D Lidar de Velodyne. Es mucho más rentable que otros de similar precio; además que conserva las características claves de los avances de Velodyne en Lidar; esto es, tiempo real, distancia 360°, 3D y mediciones de reflectividad calibrada (PUCK, 2023). Tiene las siguientes especificaciones: un peso máximo de 830 gramos, un alcance máximo de 100 metros, una precisión de ±2 centímetros, una capacidad para capturar 300,000 puntos por segundo, un campo de visión horizontal de 360 grados, un campo de visión vertical de ±15 grados, 16 haces (canales) y una tasa de rotación que varía entre 5 y 20 hercios.

3. Resultados (análisis e interpretación de los resultados)

Al analizar el algoritmo, se revela que la estimación del movimiento y la cartografía utilizando la nube de puntos de un escáner láser giratorio puede ser complicada debido a la necesidad de recuperar el movimiento y corregir la distorsión del mismo en la nube Lidar. La solución propuesta aborda este desafío al dividirlo y resolverlo a través de dos algoritmos que funcionan simultáneamente: la odometría Lidar se encarga del procesamiento más elaborado para estimar la velocidad a una frecuencia más alta, en contraste y el mapeo Lidar ejecuta un procesamiento menos complejo para crear mapas a una frecuencia más baja. La colaboración entre estos dos algoritmos permite una estimación precisa del movimiento y la generación de mapas en tiempo real.

Al implementar ese algoritmo en otro sistema operativo se obtienen resultados muy parecidos al original puesto que se identifican con éxito los puntos claves en la fase de ScanRegistration como se muestra (Figura 1) y en la fase de detección de puntos clave (Figura 2), lo cual concluye en un buena odometría (Figura 3); no obstante, en cuanto al coste computacional no se obtuvieron buenos resultados, puesto que el algoritmo fue diseñado para ejecutar cuatro procesos en paralelo y enviar mensajes por cada frame, lo que resulta complicado en C++, puesto que es necesario utilizar hilos para lograr ese objetivo; y esto concluye en un aumento en la carga de ejecución y, consecuentemente, una ralentización de la ejecución del programa.

Figura 1. Fase de Scan Registration

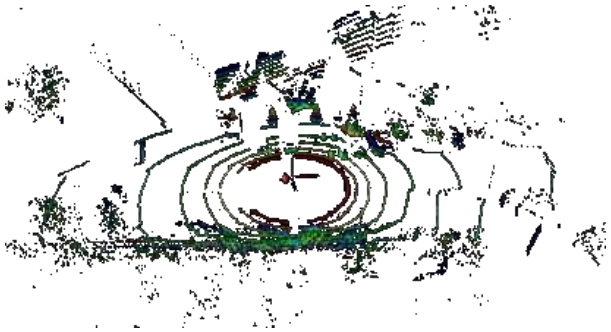
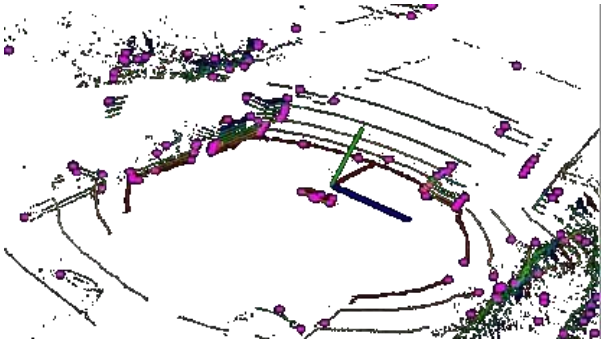
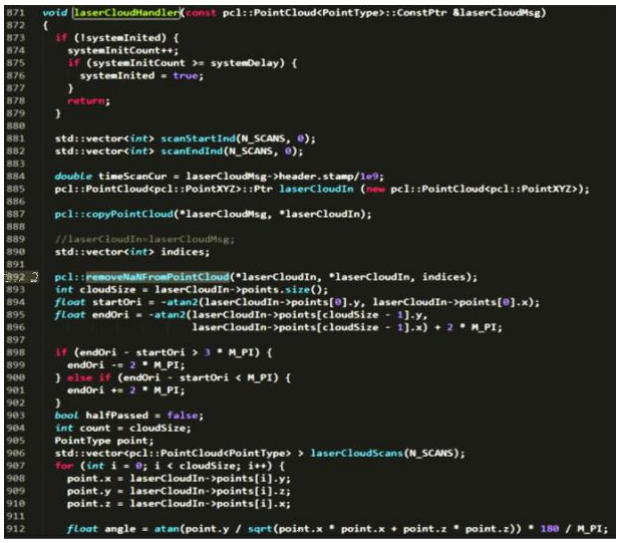


Figura 2. Puntos con forma de Esquinas detectadas



La fase de ScanRegistration incluye varias funciones, destacándose entre ellas laserCloudHandler, la cual recibe los puntos capturados por el escáner en el formato “pcl::PointCloud<PointType>”. La constante PointType equivale a pcl::PointXYZI, dependiendo del modelo del sensor Lidar utilizado.

Figura 3. Función principal de ScanRegistration

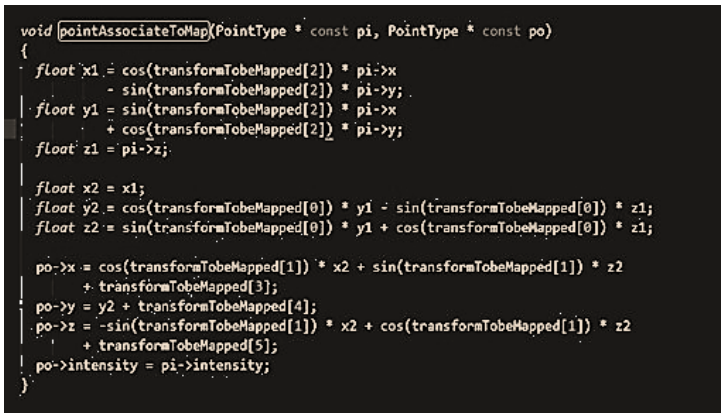


```
void laserCloudHandler(const pcl::PointCloud<PointType>::ConstPtr &laserCloudMsg)
{
  if (!systemInited) {
    systemInitCount++;
    if (systemInitCount >= systemDelay) {
      systemInited = true;
    }
    return;
  }

  std::vector<int> scanStartInd(N_SCANS, 0);
  std::vector<int> scanEndInd(N_SCANS, 0);

  double timeScanCur = laserCloudMsg->header.stamp/1e9;
  pcl::PointCloud<pcl::PointXYZ>::Ptr laserCloudIn (new pcl::PointCloud<pcl::PointXYZ>);

  pcl::copyPointCloud(*laserCloudMsg, *laserCloudIn);

  //laserCloudIn=laserCloudMsg;
  std::vector<int> indices;

  pcl::removeNaNFromPointCloud(*laserCloudIn, *laserCloudIn, indices);
  int cloudSize = laserCloudIn->points.size();
  float startOri = -atan2(laserCloudIn->points[0].y, laserCloudIn->points[0].x);
  float endOri = -atan2(laserCloudIn->points[cloudSize - 1].y,
                        laserCloudIn->points[cloudSize - 1].x) + 2 * M_PI;

  if (endOri - startOri > 3 * M_PI) {
    endOri -= 2 * M_PI;
  } else if (endOri - startOri < M_PI) {
    endOri += 2 * M_PI;
  }
  bool halfPassed = false;
  int count = cloudSize;
  PointType point;
  std::vector<pcl::PointCloud<PointType> > laserCloudScans(N_SCANS);
  for (int i = 0; i < cloudSize; i++) {
    point.x = laserCloudIn->points[i].y;
    point.y = laserCloudIn->points[i].z;
    point.z = laserCloudIn->points[i].x;

    float angle = atan(point.y / sqrt(point.x * point.x + point.z * point.z)) * 180 / M_PI;
```

Una de las funcionalidades principales es almacenar y filtrar ciertos puntos utilizando la función PCL removeNaNFromPointCloud. Esta función elimina los puntos con valores x, y o z iguales a NaN (not-a-number). La función solo calcula la correlación entre los puntos en la nube de entrada y la nube resultante después del filtrado, emitiéndola ya filtrada.

En la fase de LaserOdometry se calcula la trayectoria, la cual se mejora posteriormente en la fase de LaserMapping. Una de las funciones principales en la fase de odometría es pointAssociateToMap, que asocia los puntos clave identificados en la fase de ScanRegistration con el mapa obtenido, tal como se ilustra en la Figura 4.

Figura 4. Función de LaserOdometry

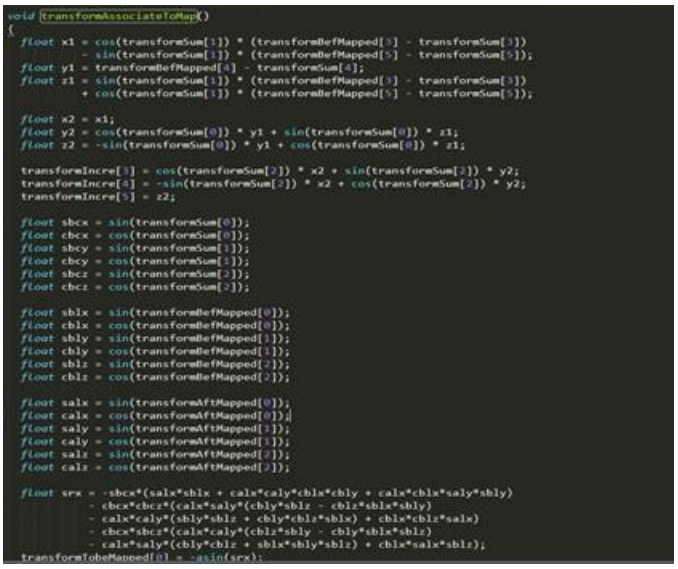


```
void pointAssociateToMap(PointType * const pi, PointType * const po)
{
  float x1 = cos(transformTobeMapped[2]) * pi->x
           - sin(transformTobeMapped[2]) * pi->y;
  float y1 = sin(transformTobeMapped[2]) * pi->x
           + cos(transformTobeMapped[2]) * pi->y;
  float z1 = pi->z;

  float x2 = x1;
  float y2 = cos(transformTobeMapped[0]) * y1 - sin(transformTobeMapped[0]) * z1;
  float z2 = sin(transformTobeMapped[0]) * y1 + cos(transformTobeMapped[0]) * z1;

  po->x = cos(transformTobeMapped[1]) * x2 + sin(transformTobeMapped[1]) * z2
        + transformTobeMapped[3];
  po->y = y2 + transformTobeMapped[4];
  po->z = -sin(transformTobeMapped[1]) * x2 + cos(transformTobeMapped[1]) * z2
        + transformTobeMapped[5];
  po->intensity = pi->intensity;
}
```

En esta función, se ingresan dos conjuntos de datos: uno contiene puntos con forma de esquinas y el otro contiene puntos planos del barrido actual, los cuales se asocian al mapa. Es relevante señalar que, en esta etapa, el algoritmo ya puede calcular la trayectoria del vehículo, aunque esta estimación todavía está en una fase inicial.

Además, en la etapa de LaserMapping se emplea una función denominada transformAssociateToMap, la cual expande la capacidad de pointAssociateToMap. Esta función utiliza los datos obtenidos de pointAssociateToMap y los transforma en puntos en los ejes x y z, tal como se ilustra en la Figura 5.

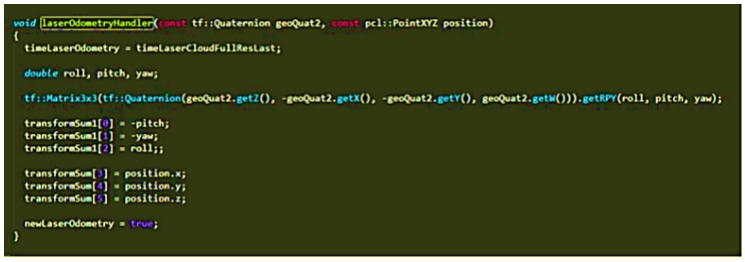
Figura 5. Función del Apartado de LaserMapping



```
void transformAssociateToMap()
{
  float x1 = cos(transformSum[1]) * (transformBefMapped[3] - transformSum[3])
           - sin(transformSum[1]) * (transformBefMapped[5] - transformSum[5]);
  float y1 = transformBefMapped[4] - transformSum[4];
  float z1 = sin(transformSum[1]) * (transformBefMapped[3] - transformSum[3])
           + cos(transformSum[1]) * (transformBefMapped[5] - transformSum[5]);

  float x2 = x1;
  float y2 = cos(transformSum[0]) * y1 + sin(transformSum[0]) * z1;
  float z2 = -sin(transformSum[0]) * y1 + cos(transformSum[0]) * z1;

  transformIncre[3] = cos(transformSum[2]) * x2 + sin(transformSum[2]) * y2;
  transformIncre[4] = -sin(transformSum[2]) * x2 + cos(transformSum[2]) * y2;
  transformIncre[5] = z2;

  float sbcx = sin(transformSum[0]);
  float cbcx = cos(transformSum[0]);
  float sbcy = sin(transformSum[1]);
  float cbcy = cos(transformSum[1]);
  float sbcz = sin(transformSum[2]);
  float cbcz = cos(transformSum[2]);

  float sblx = sin(transformBefMapped[0]);
  float cblx = cos(transformBefMapped[0]);
  float sbly = sin(transformBefMapped[1]);
  float cbly = cos(transformBefMapped[1]);
  float sblz = sin(transformBefMapped[2]);
  float cblz = cos(transformBefMapped[2]);

  float salx = sin(transformAftMapped[0]);
  float calx = cos(transformAftMapped[0]);
  float saly = sin(transformAftMapped[1]);
  float caly = cos(transformAftMapped[1]);
  float salz = sin(transformAftMapped[2]);
  float calz = cos(transformAftMapped[2]);

  float srx = -sbcx*(salx*sblx + calx*caly*cblx*cbly + calx*cblx*saly*sbly)
            - cbcx*cbcz*(calx*saly*(cbly*sblz - cblz*sblx*sbly)
            - calx*caly*(sbly*sblz + cbly*cblz*sblx) + cblx*cblz*salx)
            - cbcx*sbcz*(calx*caly*(cblz*sbly - cbly*sblx*sblz)
            - calx*saly*(cbly*cblz + sblx*sbly*sblz) + cblx*salx*sblz);
  transformTobeMapped[0] = -asin(srx);
```

Esta fase de mapeo también contribuye al cálculo matemático de la trayectoria, ya que en ella se corrige y ajusta el mapa de la trayectoria obtenido en la fase de odometría. Esto se lleva a cabo en la función laserOdometryHandler, como se ilustra en la Figura 6.

Figura 6. Función de la fase de mapeo



```
void laserOdometryHandler(const tf::Quaternion geoQuat2, const pcl::PointXYZ position)
{
  timeLaserOdometry = timeLaserCloudFullResLast;

  double roll, pitch, yaw;

  tf::Matrix3x3(tf::Quaternion(geoQuat2.getZ(), -geoQuat2.getX(), -geoQuat2.getY(), geoQuat2.getW())).getRPY(roll, pitch, yaw);

  transformSum1[0] = -pitch;
  transformSum1[1] = -yaw;
  transformSum1[2] = roll;;

  transformSum[3] = position.x;
  transformSum[4] = position.y;
  transformSum[5] = position.z;

  newLaserOdometry = true;
}
```

Este procedimiento emplea una biblioteca encontrada en la investigación bibliográfica denominada tf::Quaternion, que ejecuta rotaciones

de álgebra lineal en combinación con Matrix3x3 para calcular la posición, considerando la rotación del sensor. Esta biblioteca se desarrolló en C++, a diferencia del algoritmo original que se implementaba en C++ en conjunto con ROS.

El algoritmo hasta aquí argumentado, resulta muy eficiente, puesto que sin usar un sensor inercial se obtienen resultados válidos en cuanto al posicionamiento. Para dicha comparación fueron utilizados los datos de un GPS y se cotejaron con la ruta calculada por el algoritmo. A partir de ello, se obtuvo una similitud aceptable (Figuras 7 y 8). Ahora bien, para un análisis más profundo se calculará el error total del posicionamiento y de ciertos puntos.

Figura 7. Ruta obtenida con el algoritmo implementado

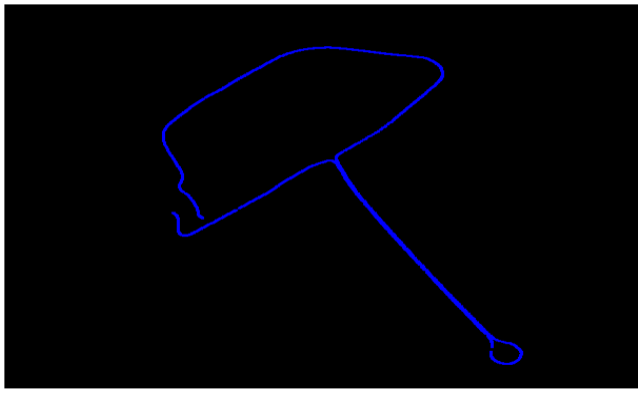


Figura 8. Ruta obtenida con el GPS

Para el respectivo análisis se definen Δx, Δy; como los errores entre la posición estimada (datos del GPS), la medida de precisión Ep se define para representar el promedio del error de posicionamiento $Ep = \frac{\sum\sqrt{\Delta x^2+\Delta y^2}}{N}$ donde N representa el alcance del escaneo, que en este experimento específico es de 1700 unidades, resultando en un error de posicionamiento total del experimento de Ep = 3.5527 metros. Además, se llevó a cabo un análisis muestral en el que se seleccionaron 6 puntos, incluyendo 3 puntos situados en una línea recta y 3 puntos en curva, como se muestra en la Tabla 1.

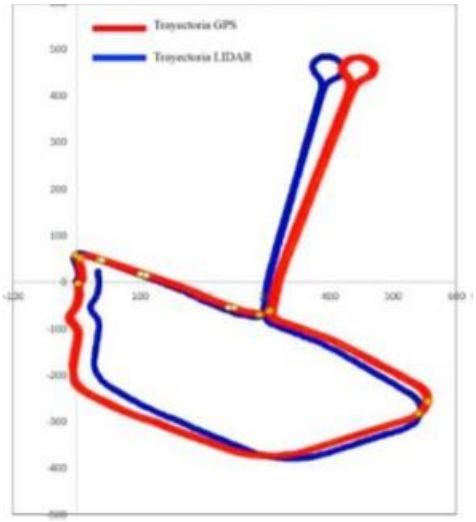
Tabla 1. Puntos en rectas y curvas

Entorno	Punto 1		Punto 2		Punto 3	
	Distancia	**Error**	**Distancia**	**Error**	**Distancia**	**Error**
Rectas	0,25 m	0,00014706	0,7 m	0,00041176	1,2 m	0,00070588
Curvas	1,50 m	0,00088235	3,2 m	0,00188235	5,6 m	0,00329412

Los puntos amarillos equivalen a las curvas y los puntos blancos a las rectas (Figura 9), donde es factible observar que el error en las curvas es

más alto que en las rectas; ello se debe con toda probabilidad a la falta de un dispositivo de medida inercial en el sistema.

Figura 9. Comparativa de las trayectorias obtenidas por el algoritmo y el GPS

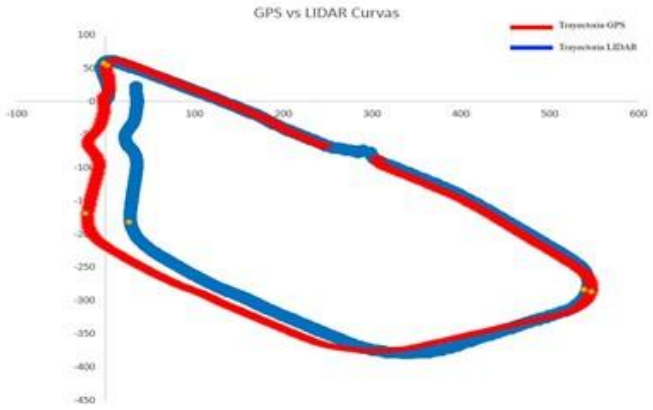


Después de examinar la trayectoria por segmentos y transformar los datos del GPS al formato X, Y, se nota que los puntos de la trayectoria generada por el algoritmo implementado no se alinean perfectamente con los datos del GPS. Por consiguiente, resulta fundamental llevar a cabo una estimación y análisis por segmentos. Esto implica extraer segmentos específicos de la trayectoria y compararlos individualmente con la trayectoria real, con el propósito de minimizar el error.

En cuanto al análisis de los segmentos curvos, se aplicó el mismo criterio de medición de error mencionado anteriormente. Para ello, se dividió el conjunto de datos mediante la extracción de la trayectoria curva, como se muestra en la Figura 10, y se comparó con el segmento obtenido del GPS. Se

obtuvo una trayectoria con un total de 898 puntos extraídos y un error Epc de 1,2 metros.

Figura 10. Trayectoria GPS versus Lidar en curvas



Se seleccionaron tres puntos para observar que, al realizar este tipo de análisis por segmentos, el error disminuye en comparación con el análisis previo, como se muestra en la Tabla 2.

Tabla 2. Puntos de segmento curvos

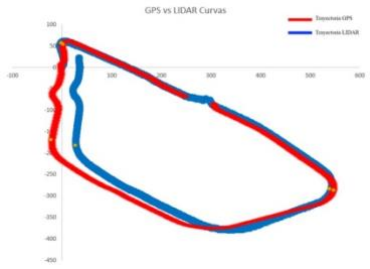
Entorno	Punto 1		Punto 2		Punto 3	
	Distancia	**Error**	**Distancia**	**Error**	**Distancia**	**Error**
Curvas	1 m	0,00111359	1,5 m	0,00244989	8,6 m	0,00957684

El error de distancia es más alto en el último punto comparado debido a que, al tratarse de una ruta algo extensa, el sistema acumula el error a lo

largo del recorrido, lo que se refleja en los extremos finales del trayecto.

Respecto al análisis de los segmentos rectos, fue utilizado el criterio de medida de error antes mencionado, para lo cual se fraccionó el conjunto de datos, extrayendo la trayectoria curva (Figura 11) y se comparó frente al segmento obtenido del GPS; lográndose una trayectoria con un total de 866 puntos extraídos y un Error Epr= 0,43 m.

Figura 11. Trayectoria GPS vs LIDAR en curvas



En este sentido, se seleccionaron tres puntos para observar que, al realizar este tipo de análisis por segmentos, se logró una considerable reducción del error en comparación con el análisis previo, como se muestra en los datos de la Tabla 3.

Tabla 3. Puntos de muestra de trayectoria rectas

Entorno	Punto 1		Punto 2		Punto 3	
	Distancia	**Error**	**Distancia**	**Error**	**Distancia**	**Error**
Rectas	0,15	0,00017321	0,35	0,00040416	0,62	0,00071594

4. Conclusiones

El empleo de bibliotecas como Eigen y tf simplifica y resuelve la necesidad de prescindir de utilizar ROS en la realización de este proyecto, ya que el algoritmo requiere el uso de ciertas bibliotecas y paquetes diseñados específicamente para ROS, las herramientas mencionadas en el texto posibilitan la implementación de las funcionalidades de dichos paquetes y bibliotecas en C++.

En la etapa de LaserOdometry, se obtiene una estimación preliminar del posicionamiento del sensor, la cual es de baja precisión. Sin embargo, en la fase de LaserMapping, se realiza una corrección y se obtiene una estimación de la posición más precisa y certera.

El algoritmo implementado muestra mayor eficiencia en tramos rectos en comparación con las curvas debido a la falta de un sensor de medida inercial. Este sensor compensa la rotación del láser en las curvas. Por esta razón, en situaciones con diversos tipos de recorrido, se recomienda realizar un análisis por segmentos para mejorar los resultados y reducir el error total del experimento.

En general, este algoritmo demuestra ser muy útil como complemento de un GPS en ciertos contextos. Sin embargo, se podría obtener información aún más precisa y detallada al incorporar un sensor IMU.

Aunque los desafíos surgieron debido a la interacción entre el entorno del recorrido y la velocidad de desplazamiento del vehículo, se sugirió realizar un conjunto más amplio de pruebas, las cuales estarían debidamente documentadas. Estas pruebas surgirían de una exhaustiva observación de todos los procesos asociados, con el objetivo de establecer una relación más precisa entre el error de la trayectoria, el tipo de entorno, la velocidad de desplazamiento del vehículo y la frecuencia de muestreo del sensor láser.

El análisis realizado puso en evidencia que la ausencia de un dispositivo de este tipo, afectaba la precisión del algoritmo; por tal razón, fue un propósito

implementar el algoritmo usando ese tipo de sensor, para realizar un tipo de fusión sensorial y para mejorar la precisión.

Entre las ventajas que se ponen de manifiesto en este artículo, como resultado de un proceso investigativo llevado a cabo durante un grupo de años, se encuentra la disponibilidad de la nube de puntos; por lo que, es hipotéticamente posible lograr la implementación de un algoritmo de detección de objetos y tracking.

5. Referencias

Eigen. (2016). *Eigen is a C++ Template Library for Linear Algebra: Matrices, Vectors, Numerical Solvers, and Related Algorithms*. http://eigen.tuxfamily.org/index.php?title=Main_Page

PCL. (2023). *About. What is PCL*. https://pointclouds.org/about/

Pinargote, V. (2017). *Implementación del algoritmo de odometría LOAM con C++ en Linux* [Tesis de Maestría, Universidad Politécnica de Madrid]. https://oa.upm.es/48115/3/TFM_VICTOR_PINARGOTE_BRAVO.pdf

PUCK. (2023). PUCK ™ (VLP-16). http://velodynelidar.com/vlp-16.html

ROS (Robot Operating System). (2021). *About ROS*. https://www.ros.org/

Tombari, F. (2013). *Keypoints and Features for 2D/3D Image Processing and Recognition* [Archivo PDF]. https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect6.pdf

Zhang, J. y Singh, S. (2014). LOAM: Lidar Odometry and Mapping in Realtime. *Robotics: Science and Systems Conference (RSS)*, p. 9. https://www.ri.cmu.edu/pub_files/2014/7/Ji_LidarMapping_RSS2014_v8.pdf