Trabajo Fin de Grado

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Guiado visuo-inercial de vehículos autónomos

Autor: Germán Ferrando del Rincón

Tutor: Carlos Vivas Venegas

Dpto. en Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Guiado visuo-inercial de vehículos autónomos

Autor:
Germán Ferrando del Rincón

Tutor:
Carlos Vivas Venegas
Profesor titular

Dpto. en Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Grado: Guiado visuo-inercial de vehículos autónomos

Autor: Germán Ferrando del Rincón

Tutor: Carlos Vivas Venegas

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

A mis padres, por haberme dado la preciosa oportunidad de desarrollarme y por haber estado siempre a mi lado en estos cuatro años.

A mis compañeros de carrera, por habernos apoyado en todo momento.

A mi tutor, Carlos, por guiarme y ayudarme a realizar este proyecto.

Germán Ferrando del Rincón

Sevilla, 2021

Resumen

La odometría es uno de los pilares en la navegación autónoma, por ello en el presente trabajo estudiaremos una técnica de odometría, la odometría visuo-inercial. En concreto la odometría visuo-inercial monocular, trataremos de usar el algoritmo VINS-Mono en un quadrotor llamado hector_quadrotor mediante simulación, estudiando la fiabilidad del algoritmo. También intentaremos realizar una fusión sensorial del algoritmo con un sensor GPS para darle mayor versatilidad y fiabilidad y por último se realizará un experimento con un control proporcional en el despegue y un vuelo paralelo al suelo. Se estudiarán gráficas para realizar comparativas en errores de la estimación de la posición.

Abstract

Odometry is clearly a pillar in autonomous navigation, therefore in this work we will study an odometry technique, visuo-inertial odometry. Specifically, monocular visuo-inertial odometry, we will try to use the VINS-Mono algorithm in a quadrotor called hector_quadrotor by simulation, studying the reliability of the algorithm. We will also try to carry out a sensory fusion of the algorithm with a GPS sensor to give it greater versatility and reliability, and finally an experiment will be carried out with a proportional control on take-off and a flight parallel to the ground. Graphs will be studied to make comparisons in errors of the estimation of the position.

Índice

ÍNDICE DE FIGURAS

Notación

UAV	Unmaned Aerial Vehicle
IMU	Inertial Measurement Unit
VINS	Visual Inertial System
VIO	Visual Inertial Odometry
VANT	Vehículo Aéreo No Tripulado
GPS	Global Positioning System
VO	Visual Odometry
ROS	Robot Operating System
RANSAC	Random Sample Consensus
PnP	Perspective n Points
GNSS	Global Navigation Satellite System
EKF	Extended Kalman Filter
KLT	Kanade-Lucas-Tomasi Feature Tracker

GFT	Good Feature To Track
SfM	Structure from Motion
URDF	Unified Robot Description Form
XML	Extensible Markup Language
MAV	Micro Aerial Vehicle
MATLAB	Matrix Laboratory
DAE	Digital Asset Exchange
XACRO	XML Macros

1 INTRODUCCIÓN

1.1 Motivación

El mundo de la robótica experimenta un auge nunca antes visto, ayudando a los humanos en todo tipo de acciones. Desde las tareas más cotidianas del día a día hasta las más complejas. Dentro de este gran y apasionante escenario han tomado recientemente un gran protagonismo los Vehículos Aéreos No Tripulados (VANT), mayormente conocidos como Unmaned Aircraft Vehicle (UAV). Sus aplicaciones son innumerables y es indudable que en el futuro abarcarán aún más.

Desde aplicaciones militares, grabación de escenas, fotogrametría, entrega de paquetería, inspección de plantas solares hasta su uso por el público en general, siendo accesibles para la mayor parte de la sociedad.
Por ello dotar a estas aeronaves de una inteligencia y autonomía mayor nos puede llevar a lograr grandes éxitos comerciales y a conseguir una mayor comodidad en la vida cotidiana de la sociedad.

Los objetivos principales en la robótica aérea se basan en mejorar la navegación autónoma y la autonomía en vuelo, nosotros nos centraremos en este proyecto en desarrollar el primero de los dos.
Para la localización de un dron en el espacio aéreo es fundamental dotarlo de sistemas de percepción, que le permitan captar información del entorno, en concreto sensores, y así poder estimar en tiempo real su posición.

Figura 1.1: Aplicación práctica de UAVs

Es imprescindible que para el desarrollo de todas las tareas descritas la intervención del humano sea la mínima, y por ello se trata continuamente de mejorar la precisión y la fiabilidad en las medidas que los sensores nos proporcionan, así como en los algoritmos desarrollados para el posicionamiento 3D del vehículo aéreo.

1.2 Objetivo

El objetivo que se persigue en este trabajo es tratar de localizar el dron en todo momento mediante una serie de sensores.

A pesar de que la tendencia natural del ser humano sería incorporar un gran número de sensores para mejorar la recogida de medidas, esto produciría un efecto indeseado ya que el diseño de un dron debe incluir un equilibrio entre funcionalidad, coste, tamaño, peso y consumo de energía. Los drones funcionan con baterías y es por ello que no es recomendable sobrecargar su consumo de energía puesto que se reduciría notablemente su autonomía.

Los sensores más comúnmente utilizados en UAVs son IMUs (Inertial Measurement Unit), encargados de medir la velocidad, la orientación y las fuerzas gravitacionales a partir de sus acelerómetros y giróscopos, los GPS, también capaces de medir velocidad y orientación, así como posición de la aeronave o cámaras a bordo, dedicadas a captar imágenes de la escena y así permitirnos obtener una representación fiel del entorno del UAV. También es común encontrar sensores térmicos, LiDAR, o de infrarrojos.

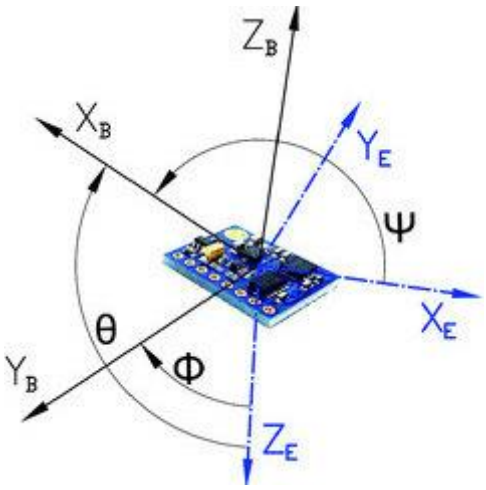


Figura 1.2 IMU con esquemático de los ángulos de Euler

En nuestro caso aplicaremos el algoritmo VINS-Mono, un robusto estimador de estado visuo-inercial, basado en dos sensores: una cámara y un IMU. El estudio VINS-Mono fue elaborado en 2018 por Tong Qin, Peiliang Li, Zhenfei Yang y Shaojie Shen pertenecientes al HUKST Aerial Robotics Group en la Universidad de Ciencia y Tecnología de Hong-Kong.

En este estudio se proponen realizar la odometría de cualquier dron basándose en dos únicos elementos, proporcionando un bajo coste en peso para el dron. La odometría visuo-inercial es una fusión de los dos sensores que se han comentado, pero no es la única manera. Es posible realizar odometría con GPS que nos daría medidas de posición absolutas, así como con múltiples cámaras.

La odometría mediante GPS es altamente precisa, pudiendo además localizar dispositivos en condiciones meteorológicas adversas. A pesar de sus ventajas el GPS puede dar medidas erróneas.

La odometría visual (VO) estudia la correspondencia entre imágenes; en concreto es el proceso mediante el cual se determina la posición y la orientación de una cámara o de un sistema de cámaras mediante el análisis de una secuencia de imágenes adquiridas, sin ningún conocimiento previo del entorno.

Como práctica general se usan varias cámaras para determinar la profundidad de la escena, lo que se conoce como visión estereoscópica. Mediante una sola imagen el ser humano no es capaz de percibir el relieve o la profundidad, pero en el caso del algoritmo VINS-Mono se resuelve este problema con una sola cámara y un

sensor inercial, lo que hace que la técnica sea óptima para drones voladores, y no tanto para vehículos terrestres ya que la autonomía juega un papel más importante en los vehículos aéreos.

VINS-Mono utiliza el mínimo conjunto de sensores posibles en tamaño, peso y potencia para la estimación de la posición de seis grados de libertad. Se utiliza un método basado en optimización no lineal para obtener una odometría visuo-inercial de alta precisión mediante la fusión sensorial de medidas preintegradas proporcionadas por la unidad inercial y detección y observación de características.

En el presente trabajo se tratará su uso en diferentes escenarios, haciendo una comparativa y analizando las características del algoritmo.

Además se tratará la fusión de la odometría recogida por el algoritmo con la de un GPS implantado en el dron, tratando de mejorar en cierta manera la precisión de las medidas.

Las simulaciones se realizarán en Gazebo, usando ROS. En concreto se usará el UAV hector_quadrotor.

Figura 1.3 Hector quadrotor en gazebo

2 ESTADO DEL ARTE

En el mundo de la navegación la odometría se basa en la utilización de la información que nos proporcionan los actuadores para interpretar un cambio de posición con la mayor precisión posible. El proceso de lectura de los sensores, almacenamiento de la información, cómputo y comunicación suponen un gasto de tiempo y recursos que se intenta minimizar al mismo. También es posible lograr la estimación de la posición a través de los sensores.

En el caso que nos concierne se realiza a través de una cámara y un sensor inercial, suponiendo un coste en peso considerablemente pequeño.

La mayoría de los artículos acerca de la odometría visual hablan sobre la odometría visual estéreo, capaz de encontrar con cierta facilidad el equivalente tridimensional a las imágenes captadas, sin embargo, la odometría con una sola cámara deberá tener algún tipo de referencia, información previa o fusión con algún otro sensor capaz de complementar su información.

En el terreno de la odometría visual con un sistema de cámaras estéreo el encargado de consolidar las bases fue Moravec, definiendo aspectos importantes como detección de características o cálculo de la estructura tridimensional.

Las aplicaciones de la odometría visual son numerosas y útiles como por ejemplo la exploración en Marte por parte del Rovers, en la cual se utilizaba un sistema parecido basado en la estimación de 6 grados de libertad y utilizando un sensor IMU para detectar cambios en roll, pitch y yaw. Presenta muchos puntos comunes con el trabajo que se va a presentar a continuación como el rechazo de valores atípicos con RANSAC pero la gran diferencia reside en que usa un sistema de cámaras estéreo.

Los sistemas que usan una odometría visual basada en una sola cámara se pueden clasificar a grandes rasgos en tres grupos: basados en características, globales e híbridos.

En los métodos basados en características el precursor fue Nister et al, que fundamentaba su trabajo en leyes geométricas para conseguir la correspondencia entre la imagen plana y una estructura tridimensional.

Los métodos categorizados como globales basan su funcionamiento en la intensidad de píxel de todos los pertenecientes a una captura. Milford y Wyeth comenzaron investigando acerca del tema, siendo sus investigaciones utilizadas posteriormente.

Los métodos globales pueden no ser todo lo robustos que se podría esperar, por ello se trató de combinar los dos métodos anteriores con el fin de mezclar las ventajas de ambos.

Actualmente los autores de VINS-Mono, utilizando un método híbrido han conseguido establecer un sistema de estimación de la posición robusto y con recuperación de fallos con una sola cámara y preintegrando las medidas de un IMU.

Los mismos, han desarrollado posteriormente un sistema llamado GVINS, capaz de combinar medidas de GNSS (Global Navigation Satellite System) con medidas visuales e inerciales.

En un futuro se podrían combinar estas técnicas con otros sensores para recabar una información más precisa o realizar un cómputo más eficiente, mediante el uso de láseres u otros sensores.

3 VINS-MONO: MARCO TEÓRICO

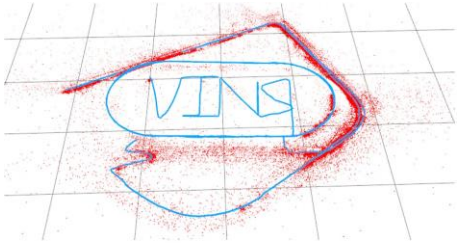
El algoritmo de VINS-Mono, como se ha comentado previamente consiste en una cámara y un IMU de bajo coste. De todos modos, la ausencia de una medida directa de posición como podría conseguirse con otro tipo de sensores tales como sensores infrarrojos, GPS u otros obligan a enfrentarse a importantes desafíos como el procesamiento del IMU, la inicialización del estimador, calibración externa y optimización no lineal.

A pesar de estos desafíos el algoritmo consigue implementar una técnica de odometría robusta para la inicialización del estimador y una relativa rápida recuperación ante errores.

El código trata de combinar de una manera efectiva la preintegración de las medidas del IMU con la observación de características.

La combinación de un módulo encargado de realizar un bucle de detección combinado con una formulación estrechamente acoplada permite la relocalización con un coste computacional mínimo lo que permite realizar la computación a bordo.

Además es posible probar su funcionamiento en dispositivos IOS sin ningún tipo de configuración, soportando el modelo de cámara pinhole, o el modelo MEI, también se puede usar para aplicaciones de realidad aumentada, aunque estas dos últimas utilidades no las usaremos en el presente trabajo.



(a) Trajectory (blue) and feature locations (red)

(b) Trajectory overlaid with Google Map for visual comparison

Figura 3.1 Comparación de VINS con Google Maps

El sistema completo puede ser satisfactoriamente aplicado en pequeños escenarios de realidad aumentada, escenarios de mediana escala para navegación de UAVs y a larga escala en tareas de estimación de estado.

Los algoritmos basados en fusión sensorial son normalmente realizados mediante un filtro de Kalman extendido (EKF), donde el IMU es usado para la propagación de estado y la visión es usada para la actualización. VINS-Mono es un algoritmo visuo-inercial estrechamente acoplado, y en este se utiliza también un EKF o un optimizador de grafos, donde las medidas de los dos sensores son optimizadas en conjunto desde un bajo nivel.

En la práctica, los sensores inerciales IMUs toman medidas como norma general a una frecuencia considerablemente más alta que las cámaras. Es por ello que se han tenido que buscar formas para manejar esta situación a la hora de combinar las medidas del IMU con las de la cámara, posteriormente serán explicadas en este proyecto.

En resumen, las contribuciones que aporta este algoritmo son:

- Un proceso de inicialización robusto, capaz de arrancar el sistema desde estados iniciales desconocidos, también es posible indicar en el algoritmo el estado inicial en aras de una inicialización más efectiva y rápida.
- Una odometría visuo-inercial monocular, basada en optimización, estrechamente acoplada con calibración extrínseca IMU-cámara.
- Bucle de detección online y relocalización estrechamente acoplada.
- Cuatro grados de libertad para la optimización del grafo de poses global.
- Demostraciones en tiempo real.

Para desgranar el complejo algoritmo se dividen sus etapas en base a este diagrama de estados.

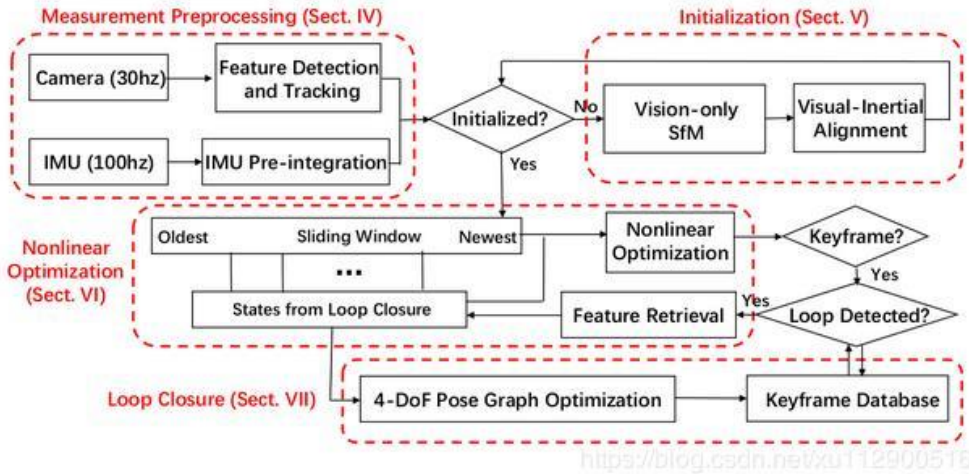


Figura 3.2 Etapas del algoritmo VINS-Mono

Se comienza a partir del bloque cuatro, en el que se explicarán los pasos de preprocesamiento para la parte visual y las medidas preintegradas del IMU, posteriormente se sigue con el desgranado del proceso de inicialización del estimador. El siguiente paso será detallar la auto calibración no lineal basada en optimización de la odometría visuo-inercial con una sola cámara y por último estudiar la relocalización y la optimización del grafo de poses global.

3.1 Preprocesamiento de medidas

Para medidas visuales, se buscan características comunes en diferentes tomas consecutivas captadas por la cámara, a partir de singularidades en las tomas y detectando nuevas características en la imagen más reciente.

Para las medidas del sensor inercial, se preintegran entre dos escenas consecutivas, es preciso tener en cuenta que al ser usado el algoritmo en un IMU de bajo coste es importante tener en cuenta el ruido y la desviación del mismo.

3.1.1 Interfaz de procesamiento de vision

Para cada nueva toma realizada, el algoritmo encargado de emparejar características es el método KLT, que está compuesto de dos pasos:

- Detección de características GFT, detector de características en la imagen basándose en entornos ricos y amplios en texturas o bordes.
- Un emparejador de características basado en el método Lucas-Kanaade.

En paralelo con el emparejamiento de características, se siguen detectando más, manteniendo un mínimo de entre 100 y 300 características por imagen. No todos los valores son utilizables, para rechazar los valores atípicos se emplea RANSAC.

Además, se seleccionan fotogramas claves, para lo cual existen dos criterios:

- Utilizando la media del paralelismo con el anterior fotograma clave, si la media de las características emparejadas entre el actual fotograma clave y el anterior está por encima de un cierto umbral se puede considerar el fotograma como nuevo fotograma clave.
- Usando el número de características emparejadas. También se estipula un umbral que en caso de ser superior al número de características emparejadas entre el fotograma clave actual y el fotograma candidato, el fotograma candidato será considerado un nuevo fotograma clave.

Este criterio está determinado con el fin de evitar una perdida completa de emparejamientos.

3.1.2 Preintegración de las medidas del IMU

Al usar un IMU de bajo coste este apartado es de vital importancia para un efectivo funcionamiento en la práctica. Al contrario que en otros trabajos desarrollados en torno a este tema, para este algoritmo se incluye una corrección del bias del IMU.

Los IMUS están formados por giroscopios y acelerómetros destinados a medir aceleraciones y velocidades.

Las leyes físicas que nos permiten caracterizar estas medidas vienen dadas por:

$$\hat{a}_t = a_t + b_{a_t} + R_w^t g^w + n_a \tag{3.1}$$
$$\hat{w}_t = w_t + b_{w_t} + n_w$$

Siendo ($\hat{w}_t$) la velocidad angular y ($\hat{a}_t$) la aceleración medidas desde el sistema de referencia del cuerpo bajo estudio.

En la implementación del código los términos de ruido $n_a \; y \; n_w$ serán tratados como nulos.

La aceleración combina mediante sumas la fuerza de la gravedad expresada en el sistema de referencia apropiado, el del cuerpo, el bias en aceleración y el ruido existente en las medidas que se asume que sigue una distribución Gaussiana $n_a \sim N(0, \sigma_a^2)$, $n_w \sim N(0, \sigma_b^2)$. El bias de la aceleración y del giroscopio vienen determinados por:

$$\dot{b}_{a_t} = n_{b_a}, \;\; \dot{b}_{w_t} = n_{b_w} \tag{3.2}$$

Dados dos instantes de tiempo correspondientes a dos imágenes b_k *y* $bk_{+1,}$ los estados de posición, velocidad y orientación pueden ser propagados como fue comentado anteriormente a partir de las medidas inerciales durante un intervalo de tiempo $[t_k, t_{k+1}]$ en el sistema de referencia global.

Quedando unas expresiones como las que siguen capaces de permitirnos calcular posición, velocidad y orientación en diferentes instantes de tiempo a partir de las medidas correspondientes a la cámara sumadas al acelerómetro y giróscopo:

$$p_{b_{k+1}}^w = p_{b_k}^w + v_{b_k}^w \Delta t_k + \iint_{t \in [t_k, t_{k+1}]} \left(R_t^w \left(\hat{a}_t - b_{a_t} - n_a\right) - g^w\right) dt^2$$

$$v_{b_{k+1}}^w = v_{b_k}^w + \int_{t \in [t_k, t_{k+1}]} \left(R_t^w \left(\hat{a}_t - b_{a_t} - n_a\right) - g^w\right) dt \tag{3.3}$$

$$q_{b_{k+1}}^w = q_{b_k}^w \otimes \int_{t \in [t_k, t_{k+1}]} \frac{1}{2} \Omega\left(\hat{w}_t - b_{w_t} - n_w\right) q_t^{b_k} dt,$$

Siendo omega dependiente de la velocidad angular, expresado como la matriz antisimétrica para producto vectorial con las componentes de w.

Δt_k es la duración del intervalo. Es visible que la propagación de estados del IMU requiere rotación, posición y velocidad de la toma de imagen b_k.

Para que el algoritmo sea robusto y eficiente es imprescindible repropagar esas medidas cuando los estados cambien, especialmente en el algoritmo basado en optimización.

Esta técnica tiene un alto gasto computacional, por este motivo se adopta un algoritmo de preintegración.

Tras cambiar el origen de referencia del mundo al de la imagen actual se pueden solo preintegrar partes

relacionadas con la aceleración lineal y la velocidad angular, pero no se repropagarán las medidas del IMU entre imágenes, lo que nos ahorra un gran gasto computacional.

$$R_w^{b_k} p_{b_{k+1}}^w = R_w^{b_k}\left(p_{b_k+}^w v_{b_k}^w \Delta t_k - \frac{1}{2} g^w {\Delta t_k}^2\right) + \alpha_{b_{k+1}}^{b_k}$$

$$R_w^{b_k} v_{b_{k+1}}^w = R_w^{b_k}\left(v_{b_k}^w - g^w \Delta t_k\right) + \beta_{b_{k+1}}^{b_k} \qquad (3.4)$$

$$q_{b_k}^w \otimes q_{b_{k+1}}^w = \gamma_{b_{k+1}}^{b_k}$$

Donde alpha, gamma y beta son términos que representan los términos de aceleración que se integran para contribuir al estudio de la posición, velocidad y orientación.

$$\alpha_{b_{k+1}}^{b_k} = \iint_{t\in[t_k,t_{k+1}]} (R_t^{b_k}\left(\hat{a}_t - b_{a_t} - n_a\right) dt^2$$

$$\beta_{b_{k+1}}^{b_k} = \int_{t\in[t_k,t_{k+1}]} (R_t^{b_k}\left(\hat{a}_t - b_{a_t} - n_a\right) dt \qquad (3.5)$$

$$\gamma_{b_{k+1}}^{b_k} = \int_{t\in[t_k,t_{k+1}]} \frac{1}{2}\Omega\left(\widehat{w}_t - b_{w_t} - n_w\right)\gamma_t^{b_k} dt$$

Estos términos se conocen como los términos de preintegración, y se pueden obtener solo con las medidas del IMU, no es necesario la actuación de la cámara si tenemos en cuente como origen de referencias la imagen actual b_k ya que los términos indicados solo se refieren al bias del IMU, si existe un cambio pequeño en este podemos estimar estos términos a partir de aproximaciones de primer orden.

En caso de ser grandes cambios será necesario hacer repropagación.

Para su implementación discreta será necesario aplicar un método de integración numérica diferencial.

Tras aplicarlo a los anteriores términos de preintegración teniendo en cuenta la matriz de covarianza podemos escribir la ecuación matricial resultante que nos permitirá implementar el algoritmo:

$$\begin{bmatrix} \hat{\boldsymbol{\alpha}}_{b_{k+1}}^{b_k} \\ \hat{\boldsymbol{\beta}}_{b_{k+1}}^{b_k} \\ \hat{\boldsymbol{\gamma}}_{b_{k+1}}^{b_k} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \mathbf{R}_w^{b_k}(\mathbf{p}_{b_{k+1}}^w - \mathbf{p}_{b_k}^w + \frac{1}{2}\mathbf{g}^w \Delta t_k^2 - \mathbf{v}_{b_k}^w \Delta t_k) \\ \mathbf{R}_w^{b_k}(\mathbf{v}_{b_{k+1}}^w + \mathbf{g}^w \Delta t_k - \mathbf{v}_{b_k}^w) \\ \mathbf{q}_{b_k}^{w^{-1}} \otimes \mathbf{q}_{b_{k+1}}^w \\ \mathbf{b}_{a b_{k+1}} - \mathbf{b}_{a b_k} \\ \mathbf{b}_{w b_{k+1}} - \mathbf{b}_{w b_k} \end{bmatrix}$$

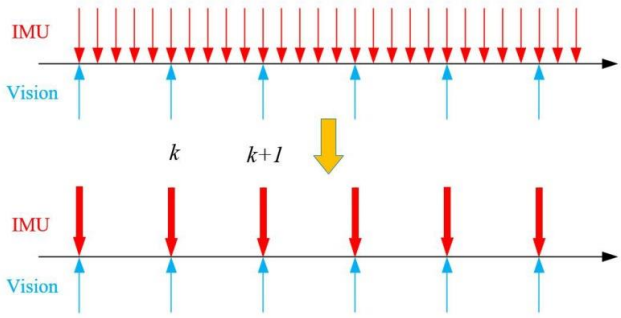


Figura 3.3 Preintegración del IMU

3.2 Inicialización del estimador

En la odometría visuo-inercial que estamos estudiando la escala no es apreciable directamente, ya que solo poseemos una cámara.

Debido a este aspecto no es tarea sencilla conseguir fusionar las medidas de dos sensores sin unos valores iniciales suficientemente buenos. La inicialización es probablemente la etapa menos robusta en el algoritmo.

Para conseguir una inicialización apropiada se acude a solo visión, SLAM, o SfM, Structure from Motion. En la gran mayoría de ocasiones solo con visión es posible comenzar a funcionar basándose en métodos como el algoritmo de ocho puntos.

La inicialización es dividida en dos etapas que describiremos a continuación:

3.2.1 Ventana deslizante, solo visión SfM

Se produce el comienzo de la inicialización solo con visión para estimar un grafo a escala de la pose de la cámara.

Se usa un cierto límite de imágenes, el conjunto de las imágenes cambia constantemente y es llamado ventana deslizante, para no incrementar en exceso el gasto computacional, en un primer lugar se observan características parecidas entre la última imagen y las anteriores. Cuando se consigue la suficiente semejanza entre la actual y cualquier otra perteneciente a la ventana deslizante se usa el algoritmo de cinco puntos para obtener la rotación relativa entre esos sistemas de referencia y la traslación.

En caso de que el algoritmo sea exitoso triangulamos todas las características entre esas dos imágenes, y con esa triangulación se estiman las poses de las demás imágenes pertenecientes a la ventana.

Al no tener todavía un sistema de referencia global mundo debemos referenciarlo a alguno conocido, se escoge el sistema de referencia de la cámara.

3.2.2 Alineación visuo-inercial

1) Calibración del sesgo (bias) del giroscopio: Si escogemos dos imágenes consecutivas pertenecientes a la Ventana deslizante, obtenemos la rotación respecto al sistema de referencia escogido (el de la cámara) junto con el parámetro gamma de la preintegración del IMU obtenido en el apartado 2.1 linealizamos respecto al sesgo del giroscopio, al minimizar la función de coste resultante:

$$\min_{\delta b_w} \sum_{k\in\beta} ||q_{b_{k+1}}^{c0}{}^{-1} \otimes q_{b_k}^{c0} \otimes \gamma_{b_{k+1}}^{b_k}||^2 \quad (3.7)$$

β comprende todas las imágenes de la Ventana deslizante. De esta forma conseguimos una aproximación del parámetro gamma en relación con el sesgo del giroscopio.

2) Velocidad, vector de gravedad e inicialización de escala métrica: Una vez inicializado el giroscopio se trata de inicializar parámetros importantes para la navegación.

$$X_I = [v_{b_o}^{b_o}, v_{b_1}^{b_1}, \ldots . v_{b_n}^{b_n}, g^{c_o}, s] \quad (3.8)$$

El objetivo es tener todos los datos de este vector que contiene todos los parámetros previamente descritos, las respectivas velocidades se refieren a las velocidades en el instante de tomar la foto según su índice respecto al sistema de referencia del cuerpo bajo estudio.

Si reformulamos la ecuación 2.4 y lo combinamos con el desarrollo de la calibración del giroscopio podemos obtener una ecuación a minimizar que nos dará como resultado las velocidades para cada imagen de la ventana, así como los parámetros pertenecientes al vector previamente descrito.

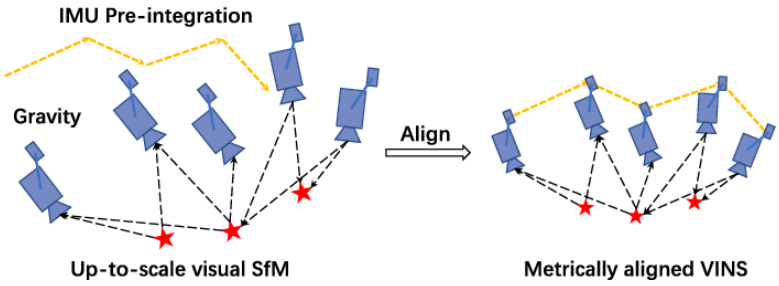


Figura 3.4 Alineación visuo-inercial VINS-Mono

3) Refinamiento de la gravedad: Es posible tras el paso dos conseguir reducir a dos grados de libertar el vector de gravedad. Se expresa el vector de gravedad como el valor conocido de la gravedad multiplicado por un vector unitario representative de la dirección de la gravedad y sumado con dos vectores ortogonales al plano tangente a la esfera unitaria que se forma de recorrer todos los posibles ángulos con el vector de gravedad.

$$g.\hat{\boldsymbol{g}} + w_1 b_1 + w_2 b_2 \quad (3.9)$$

Los parámetros omegas se corresponden al desplazamiento a través de b_1 y b_2.

El proceso para hallar los vectores ortogonales es fácilmente explicable mediante un sencillo pseudo-código:

SI $\hat{\boldsymbol{g}} \neq [1,0,0]$ $entonces$

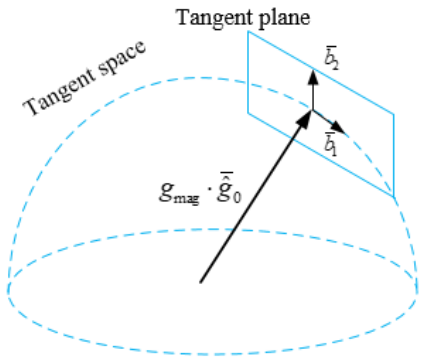
$b_1 = \frac{\hat{\boldsymbol{g}} x [1,0,0]}{||\hat{\boldsymbol{g}} x [1,0,0]||}$;

En caso contrario

$b_1 = \frac{\hat{\boldsymbol{g}} x [0,0,1]}{||\hat{\boldsymbol{g}} x [0,0,1]||}$;

Fin Si

$b_2 = \hat{\boldsymbol{g}} x b_1$;



Esfera unitaria del vector de gravedad

Figura 3.5 Representación gráfica del refinamiento de la gravedad

4) Completar la inicialización: Una vez refinado el vector de gravedad deberemos obtener la rotación entre el mundo y la cámara. Una vez obtenida con la rotación de la gravedad al eje z rotamos todas las variables para expresarlas en el sistema de referencia del mundo.

3.3 Odometría visuo-inercial monocular estrechamente acoplada

Tras la inicialización se comienza con la odometría. La odometría estará basada en la ventana deslizante.

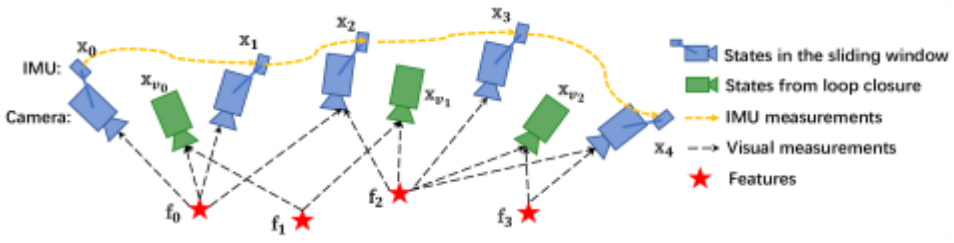


Figura 3.6 Concepto de odometría VINS-MONO

3.3.1 Formulación

En este apartado se detalla la ecuación a ser minimizada para obtener la mejor estimación. Se crea un vector tal que:

$$\begin{aligned} X &= [x_0, x_1, \ldots . x_n, x_c^b, \lambda_0, \lambda_1, \ldots . . \lambda_m] \\ x_k &= [p_{b_k}^w, v_{b_k}^w, q_{b_k}^w, b_a, b_g], k \in [0, n] \\ x_c^b &= [p_c^b, q_c^b], \end{aligned} \tag{3.10}$$

Donde el primer vector está compuesto por:

-Los vectores x que representan el estado del IMU en la imagen capturada k, contienen posición, velocidad, orientación y los términos correspondientes al sesgo.

-El parámetro n se refiere al número de fotogramas claves y m es el número total de características en la ventana deslizante.

-Lambda de uno será por ejemplo la inversa de la profundidad de la característica l-ésima desde su primera observación.

El algoritmo se encargará de minimizar la suma de lo anterior junto con la norma de Mahalanobis de los residuos de todas las medidas. La ecuación resultante quedará de la siguiente forma:

$$\min_{X} \left\{ ||r_p - H_p X||^2 + \sum_{k \in B} || r_b(\hat{z}_{b_{k+1}}^{b_k}, X)||_{P_{b_{k+1}}^{b_k}}^2 + \sum_{(l,j) \in C} \rho || r_c(\hat{z}_l^{c_j}, X)||_{P_l^{c_j}}^2 \right\} \tag{3.11}$$

La norma de Huber queda detallada como:

$$\rho(s) = \begin{cases} 1, & s \geq 1 \\ 2\sqrt{s} - 1, & s < 1 \end{cases}$$

Los residuos de las medidas del IMU son denotados por el subíndice b y los residuos visuales por el c.

$$\begin{matrix} r_b(\hat{z}_{b_{k+1}}^{b_k}, X) \\ r_c\left(\hat{z}_l^{c_j}, X\right) \end{matrix} \tag{3.12}$$

Para resolver esta ecuación no lineal es necesaria una herramienta como Ceres Solver.

3.3.2 Medida residual del IMU

El residuo del IMU se puede expresar como:

$$\begin{aligned} \mathbf{r}_{\mathcal{B}}(\hat{\mathbf{z}}_{b_{k+1}}^{b_k}, \mathcal{X}) &= \begin{bmatrix} \delta\boldsymbol{\alpha}_{b_{k+1}}^{b_k} \\ \delta\boldsymbol{\beta}_{b_{k+1}}^{b_k} \\ \delta\boldsymbol{\theta}_{b_{k+1}}^{b_k} \\ \delta\mathbf{b}_a \\ \delta\mathbf{b}_g \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{R}_w^{b_k}(\mathbf{p}_{b_{k+1}}^w - \mathbf{p}_{b_k}^w + \frac{1}{2}\mathbf{g}^w\Delta t_k^2 - \mathbf{v}_{b_k}^w\Delta t_k) - \hat{\boldsymbol{\alpha}}_{b_{k+1}}^{b_k} \\ \mathbf{R}_w^{b_k}(\mathbf{v}_{b_{k+1}}^w + \mathbf{g}^w\Delta t_k - \mathbf{v}_{b_k}^w) - \hat{\boldsymbol{\beta}}_{b_{k+1}}^{b_k} \\ 2\left[\mathbf{q}_{b_k}^{w^{-1}} \otimes \mathbf{q}_{b_{k+1}}^w \otimes (\hat{\boldsymbol{\gamma}}_{b_{k+1}}^{b_k})^{-1}\right]_{xyz} \\ \mathbf{b}_{ab_{k+1}} - \mathbf{b}_{ab_k} \\ \mathbf{b}_{wb_{k+1}} - \mathbf{b}_{wb_k} \end{bmatrix} \end{aligned} \tag{3.13}$$

3.3.3 Medida residual de visión

El residuo de la medida de la cámara es definido en una esfera unitaria al contrario que en el modelo pinhole.

Considerando la l-ésima característica que es por primera vez observada en la imagen i-ésima el residuo para la observación de características en la imagen j-ésima es definido como:

$$\begin{aligned} \mathbf{r}_{\mathcal{C}}(\hat{\mathbf{z}}_l^{c_j}, \mathcal{X}) &= \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 \end{bmatrix}^T \cdot (\hat{\bar{\mathcal{P}}}_l^{c_j} - \frac{\mathcal{P}_l^{c_j}}{\|\mathcal{P}_l^{c_j}\|}) \\ \hat{\bar{\mathcal{P}}}_l^{c_j} &= \pi_c^{-1}(\begin{bmatrix} \hat{u}_l^{c_j} \\ \hat{v}_l^{c_j} \end{bmatrix}) \\ \mathcal{P}_l^{c_j} &= \mathbf{R}_b^c(\mathbf{R}_w^{b_j}(\mathbf{R}_{b_i}^w(\mathbf{R}_c^b\frac{1}{\lambda_l}\pi_c^{-1}(\begin{bmatrix} u_l^{c_i} \\ v_l^{c_i} \end{bmatrix}) \\ &\quad + \mathbf{p}_c^b) + \mathbf{p}_{b_i}^w - \mathbf{p}_{b_j}^w) - \mathbf{p}_c^b), \end{aligned} \tag{3.14}$$

La elección de los vectores ortogonales ha sido descrita previamente mediante pseudo-código.

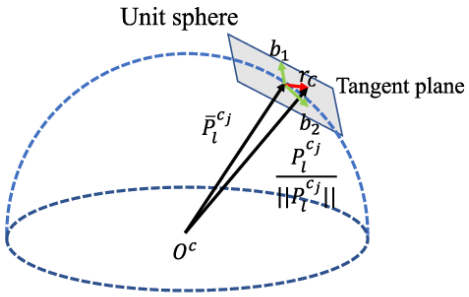


Figura 3.7 Residuo visual en esfera unitaria

3.3.4 Marginalización

La marginalización está pensada fundamentalmente para reducir el gasto computacional del algoritmo.

Se trata de escoger que imágenes han de permanecer en la ventana deslizante y cuales no, dependiendo de si se trata de un fotograma clave o no. No todos los fotogramas claves son marginalizados, el objetivo es mantener los fotogramas espacialmente separados en la ventana.

3.3.5 Ajuste del paquete visuo-inercial para estimación de estado

Para ajustar la frecuencia de computación a la frecuencia de captación de imágenes en dispositivos con bajo poder computacional se emplea un ligero paquete para ajustar la estimación de estado.

La función de coste es la misma que en el apartado 2.3.1, pero en cambio solo se optimizan las poses y velocidades de un cierto número de estados previos del IMU.

3.3.6 Detección de fallos y recuperación

A pesar de ser el algoritmo robusto ante entornos difíciles y cambios bruscos en movimiento los fallos son algo inevitable. Para detectar fallos se usan unos ciertos estándares que describiré a continuación:

- El número de características observadas en el último fotograma es menor que un cierto umbral
- Una larga discontinuidad en posición o rotación entre las dos últimas salidas del estimador
- Variaciones grandes en el sesgo o en los parámetros extrínsecos.

Cuando se detecta un fallo, se retoma la inicialización, creando un nuevo grafo de poses.

3.4 Relocalización

A pesar de la importancia de la marginalización para la reducción del gasto computacional ésta también introduce derivas en la estimación. En concreto la deriva ocurre en la posición tridimensional y en la rotación en torno a la gravedad (yaw).

Se precisan tres etapas para la relocalización:

3.4.1 Detección de bucle

Para la detección de bucles se utiliza DBoW2, que es un proyecto de código abierto escrito en C++ que calcula de forma veloz las similitudes entre dos fotogramas.

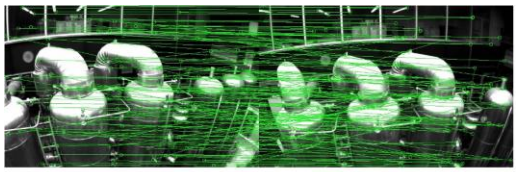
Adicionalmente a las esquinas usadas para la odometría visuo-inercial, se incluyen 500 más gracias al descriptor BRIEF, lo que permite mejorar la detección de bucle.

En general la detección de bucle nos permite a partir de los datos almacenados conocer si nos encontramos en una situación parecida.

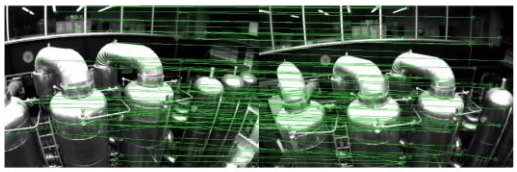
3.4.2 Recuperación de características

Una vez detectado un bucle, gracias al descriptor BRIEF conseguimos corresponder las imágenes de la ventana deslizante y el candidato identificado por la detección de bucle. Es común que haya bastantes valores atípicos, por ello se usan dos pasos geométricos para rechazarlos que son:

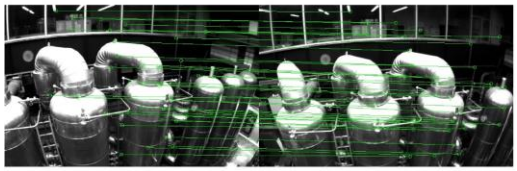
1) 2D-2D: Un test de la matriz fundamental con RANSAC
2) 3D-2D: Test PnP con RANSAC, basándose en la posición 3D de las características contenidas en la ventana deslizante y las observaciones 2D en el candidato indicado por la detección de bucle se ejecuta un test PnP.



(a) BRIEF descriptor matching results



(b) First step: 2D-2D outlier rejection results



(c) Second step: 3D-2D outlier rejection results.

Figura 3.8 Expulsión de valores atípicos con RANSAC

3.4.3 Relocalización estrechamente acoplada

La relocalización nos permite alinear la ventana deslizante actual con el grafo de antiguas poses.

Tras estos pasos podemos modificar la función de coste añadiéndole los términos adicionales correspondientes al bucle.

$$\min\left\{\left|\left|r_p - H_p X\right|\right|^2 + \sum_{k\in B}\left|\left| r_b\left(\hat{z}_{b_{k+1}}^{b_k}, X\right)\right|\right|_{P_{b_{k+1}}^{b_k}}^{2} + \sum_{(l,j)\in C}\rho\left|\left| r_c\left(\hat{z}_l^{c_j}, X\right)\right|\right|_{P_l^{c_j}}^{2} + \sum_{(l,v)\in C}\rho\left|\left| r_c(\hat{z}_l^v, X, \hat{q}_v^w, \hat{p}_v^w)\right|\right|_{P_l^{c_j}}^{2}\right\} \quad (3.15)$$

Donde (l,v) significa la l-ésima característica observada en el cierre de bucle de la imagen v.

3.5 Optimización del grafo de poses global

Una vez completada la relocalización, la ventana global se alinea con las respectivas poses antiguas, este paso se encarga de asegurar que las poses antiguas son registradas y guardadas en una configuración global.

Como se ha comentado previamente la deriva solo ocurre en cuatro de los seis grados de libertad posibles, es por esta razón que ignoraremos estimar los estados de roll y pitch.

Cuando un fotograma clave es marginalizado fuera de la ventana deslizante se añade al grafo de poses sirviendo como vértice, es decir, será útil para ser un nexo de conexión con otros vértices.

Posteriormente se optimizan los cuatro grados de libertad que incluyen deriva.

La optimización y la relocalización son ejecutadas en dos hilos diferentes, en paralelo, lo que permite a la relocalización hacer uso del grafo de poses más optimizado en cuanto esté disponible.

Es considerable la magnitud a la que puede llegar el grafo de poses, ya que crece sin control cuando la distancia recorrida por el dron aumenta. Para solucionarlo se emplea un proceso de reducción de muestras en aras de mantener la base de datos a un tamaño limitado.

4 IMPLEMENTACIÓN: ROS Y GAZEBO

Todos los experimentos, simulaciones y algoritmos han sido desarrollado en ROS (Robot Operating System), ROS es un framework flexible utilizado para escribir software relacionado con robots.
Es una colección de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto en una amplia variedad de escenarios y plataformas.

ROS es un framework disponible gratuitamente para el público y contiene un gran abanico de posibilidades y librerías principalmente desarrolladas en C++ y Python, formando así un esqueleto modular y escalado.

Su estructura está compuesta fundamentalmente por nodos, que constituyen un proceso encargado de realizar una tarea concreta, desde el lanzamiento y contenido de una simulación hasta un complejo cálculo.

ROS y la robótica colaborativa

Figura 4.1 Ros y su entorno

Los nodos previamente descritos interaccionan entre ellos a través de topics, que son los canales que usan los nodos para intercambiar mensajes entre ellos.
Si tuviésemos un nodo encargado de leer por ejemplo las coordenadas GPS de nuestro dron y otro para convertirlas a coordenadas cartesianas, éstos deberán comunicarse a través de un topic, que actúa de enlace entre los dos nodos.
En concreto podemos subscribirnos a un topic si queremos acceder a la información que circula dentro suya o

podremos publicar si nuestro interés reside en introducir una cierta información por el canal
¿Y qué se intercambia entre los nodos y por ende a través de los topics? La respuesta es sencilla, mensajes.

Los mensajes son un conjunto de datos estructurados de una determinada manera, los mensajes pueden tener estructuras predeterminadas ya creadas o podemos crearlos nosotros mismos ajustándolos a nuestras necesidades particulares.

En concreto ROS se subdivide en dos partes: el sistema operativo, ros, y ros-pkg, los paquetes integrados que surgen a partir de la colaboración y contribución de los propios usuarios y que desarrollan tareas específicas como localización, planificación percepción, etc.

4.1 Sistema operativo

El algoritmo VINS-Mono está desarrollado en ROS Kinetic, una versión de ROS que se corresponde mayormente a Ubuntu 16.04. En un principio se trató de correr el algoritmo en Ubuntu 16.04, pero la complicación de configuración de los drivers en el ordenador y los problemas constantes obligaron a traspasar el proyecto a Ubuntu 18.04 y ROS Melodic.

ROS Melodic esta principalmente orientado para ser usado en Ubuntu 18.04 aunque también puede ser utilizado en Ubuntu 17.0 y Debian Stretch

Para instalar Ubuntu es preciso realizar una partición de disco duro para compartir espacio entre Ubuntu y el sistema operativo principal, Windows.

4.2 Funcionalidades

Existen una amplia gama de herramientas y funcionalidades que permiten al desarrollador realizar tareas complejas, como son:

1) RVIZ: Se trata de un entorno de simulación y visualización tridimensional para entornos robóticos.

En nuestro caso lo usaremos para visualizar en tiempo real la odometría realizada por el algoritmo VINS-Mono.

2) Rosbag: Mediante la terminal podemos utilizar esta herramienta para guardar y reproducir posteriormente el contenido que circula a través de los topics.

Rosbag utiliza un determinado formato, el .bag, que registra los mensajes de un topic y los guarda con un registro temporal.

En el presente trabajo se usará con el fin de mostrar posteriormente resultados en MATLAB.

3) Catkin: Catkin es la herramienta de compilación de ROS, tiene su origen en CMake y es open source e independiente del lenguaje de programación.

4) Roslaunch: Es una herramienta que permite al desarrollador lanzar uno o más nodos a la vez, es muy útil debido a que nos permite configurar múltiples parámetros de posterior aplicación práctica.

El lenguaje de programación que utiliza Es XML.

Además de estas herramientas existen muchas otras que nos dan un amplio repertorio de posibilidades para

interactuar con el mundo de la robótica ya sea mediante simulación o con el hardware real.

4.3 Gazebo

Gazebo es una plataforma de uso libre para simulación en tres dimensiones que tiene muchas características provechosas para el desarrollador como:

1) Simulación dinámica con numerosos motores de física de alto rendimiento.
2) Gráficos tridimensionales muy desarrollados para poder generar entornos con un alto componente real renderizando texturas, luces, etc.
3) Sensores de los que podemos obtener datos desde la simulación
4) Plugins para poder crear distintos robots, acciones o los mismos sensores

En concreto se ha utilizado la versión de Gazebo 9.

Figura 4.2 Simulador 3D Gazebo

4.4 Dron Hector_Quadrotor

Para probar el algoritmo se necesita un vehículo móvil, capaz de excitar el IMU lo suficiente para que el algoritmo VINS-Mono funcione correctamente, en un principio se probaron varios robots terrestres, incluso echando a correr el algoritmo en uno de ellos, usando el paquete omni3ros_pkg, un robot omnidireccional de tres ruedas, pero finalmente se descartó la idea ya que las aplicaciones eran mucho mayores si conseguíamos

correr el algoritmo en un dron volador.

En un principio Hector_Quadrotor está diseñado para Ros Kinetic, lo que dificultó enormemente la tarea, teniendo muchos problemas para su instalación, cambiando incluso la herramienta Qt de Linux para poder adaptarlo al paquete especificado junto con más modificaciones.

El paquete fue desarrollado por Stefan Kohlbrecher y Johan Meyer, y contiene los paquetes necesarios para la simulación de un vehículo aéreo no tripulado de cuatro motores tales como la descripción del quadrotor que contiene los archivos URDF, que son archivos en formato XML destinados a representar el modelo del robot.

El paquete interno hector_quadrotor_description contiene múltiples apartados en los que encontramos todos los sensores posibles para el UAV, pudiendo usarse una parte de ellos o todos.

También se incluye un nodo de teleoperación que usaremos para controlar el dron remotamente mediante teclado durante los experimentos, así como controladores, plugins, representación en gazebo, mensajes específicos y demostraciones.

Este paquete ha sido clonado en el entorno de trabajo desde github.

5 ARRANQUE DE VINS-MONO

VINS-Mono es un algoritmo versátil que se puede arrancar tanto en simulación como en experimentos físicos, en el presente proyecto se simulará en el dron hector_quadrotor.

5.1 Descarga y prueba con los datasets públicos

El primer paso para ejecutar con éxito el algoritmo será descargar su código de github y compilarlo.

Antes de la instalación del algoritmo en sí, comenzaremos por instalar los paquetes necesarios para el correcto funcionamiento del paquete que serán:

A) Ceres Solver: Es una librería de código abierto en C++ utilizada para modelar y resolver grandes problemas de optimización, en el algoritmo es necesario su uso para resolver las funciones de coste expresadas en el apartado 2.3.1

B) OpenCV 2.4.9: A pesar de que en la página de github de los autores originales expresan que se resolvió con la versión 3.3.1 en nuestro caso no funcionaba por lo que hubo que probar con todas las versiones posibles hasta que hubo una con la que se consiguió implementarlo eficazmente

C) Eigen 3.3.3: Este paquete contiene funciones para el cálculo simbólico de valores y vectores propios.

Una vez conseguido instalar correctamente todas estas librerías y paquetes se procede a la instalación en la que básicamente se clona el repositorio de github y se procede a compilar el código.

Código 5.1 Instalación de VINS-Mono desde la terminal

```
cd ~/<catkin_ws>/src
git clone https://github.com/HKUST-Aerial-Robotics/VINS-Mono.git
cd ..
catkin_make
source ~/<nombre_ws>/devel/setup.bash
```

Los autores ponen a nuestra disposición una serie de dataset públicos, EUROC MAV datasets, que han sido recopilados a bordo de un micro vehículo aéreo (MAV). Aunque los datasets son recopilados con un sistema de cámaras estéreo sólo se usará una de ellas en las pruebas. La cámara tiene un framerate de 20 FPS y el sensor IMU que es un MEMS IMU se refresca a una frecuencia de 200 Hz.

Además, se muestran en los datasets el ground_truth para comparar después con el resultado estimado y llegar a conclusiones.

El vehículo usado para la recopilación de datos es un helicóptero con rotor hexagonal y sensor visuo-inercial.

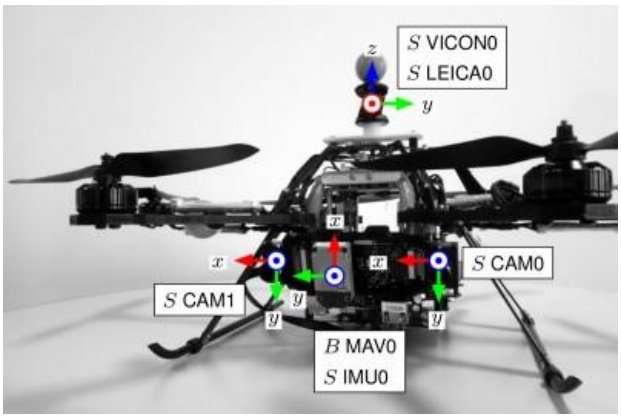


Figura 5.1 Vehículo usado para guardar los datsets

Existen doce datasets, uno para calibración de parámetros extrínsecos, y los demás para comprobar la fiabilidad del algoritmo en diferentes niveles de dificultad. La dificultad aumenta progresivamente desde el dataset 1 hasta el 11.

Todas las medidas contenidas en estos datos están referidas en el sistema internacional excepto el tiempo, expresado en nanosegundos. Asimismo, todas las medidas de los sensores están referenciadas a su propio sistema de referencia.

En este apartado procederemos a comprobar cómo responde el algoritmo ante distintos niveles de dificultad, es importante tener en cuenta que el archivo de configuración euroc.yaml, que será el necesario para en posteriores experimentos configurar correctamente el arranque del algoritmo en distintos drones, está perfectamente configurado para ser simulado con los datasets.

Para realizar una comparativa e ilustrar gráficamente la respuesta del algoritmo ante los datasets nos aprovecharemos de que en el propio paquete de VINS-Mono existe una opción para publicar y visualizar ground truth para cada uno de los datasets.

De esta manera podremos guardar en dos archivos .bag la odometría de VINS-Mono y ground truth para su posterior análisis.

Para visualizar por ejemplo el primer dataset deberemos ejecutar los siguientes comandos:

Código 5.2 Comandos para experimento con el primer dataset

```
roslaunch vins_estimator euroc.launch
roslaunch vins_estimator vins_rviz.launch
rosbag record -O vins /vins_estimator/odometry
rosbag record -O gt /benchmark_publisher/odometry
rosbag play MH_01_easy.bag
roslaunch benchmark_publisher publish.launch  sequence_name:=MH_01_easy
```

De esta manera tendremos dos archivos .bag que interpretaremos posteriormente en MATLAB

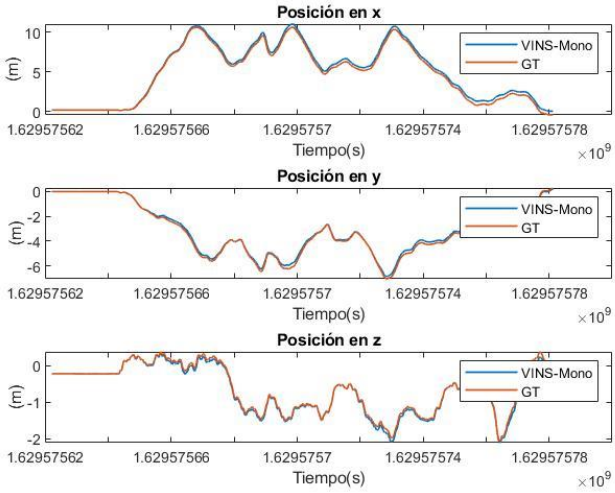


Figura 5.2 Evolución MH01

Como podemos observar funciona realmente bien, obteniendo unos resultados óptimos en la trayectoria más sencilla.

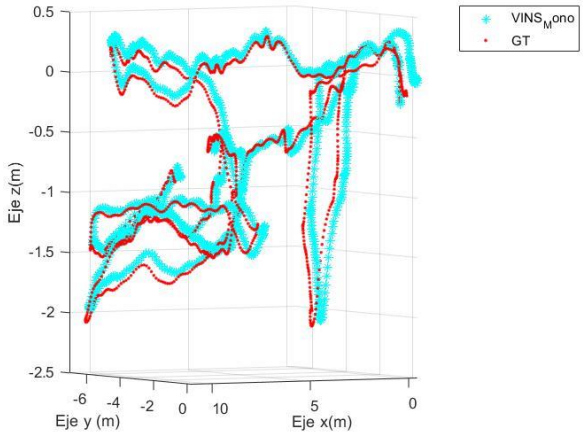


Figura 5.3 Trayectoria tridimensional MH01

Y, como última gráfica para cuantificar el error, ya que a simple vista es difícil distinguir cuando hay tantas similitudes en las dos gráficas, una gráfica que representa el error cuadrático en cada instante:

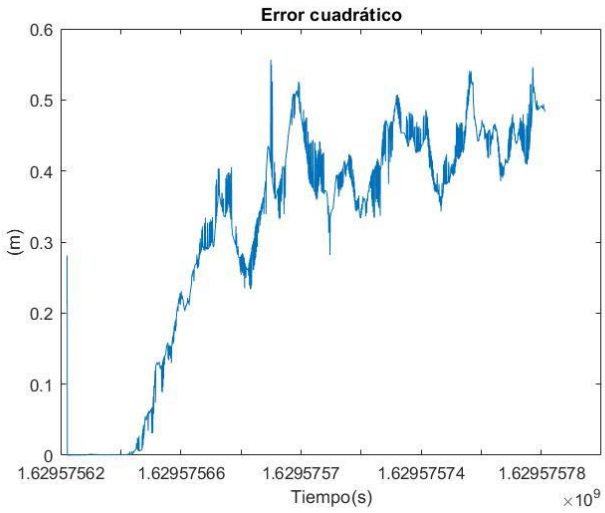


Figura 5.4 Error cuadrático MH01

Como podemos observar el error es mínimo, ni siquiera alcanzando el metro, lo que se puede considerar una buena estimación de la posición en el ámbito de los drones.

A continuación, procedemos a guardar la trayectoria realizada poniendo en pose_graph_save_path el directorio donde deseamos almacenar la información relativa a dicha trayectoria, el algoritmo corregirá errores en la estimación usando los datos que previamente hemos guardado en dicho directorio, fotogramas clave y datos de la navegación.

Una vez realizado la captura se procede a poner el parámetro load_previous_pose_graph a 1 conseguiremos que en el siguiente experimento use la referencia previamente guardada.

Los resultados se obtienen con el mismo procedimiento que para la primera repetición.

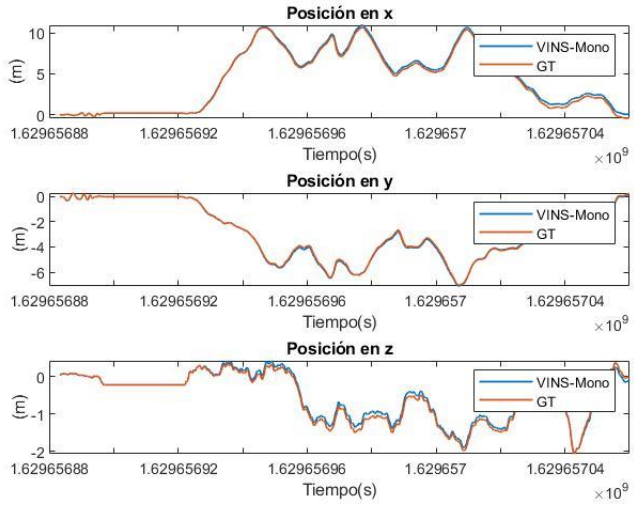


Figura 5.5 Evolución con trayectoria guardada de MH01

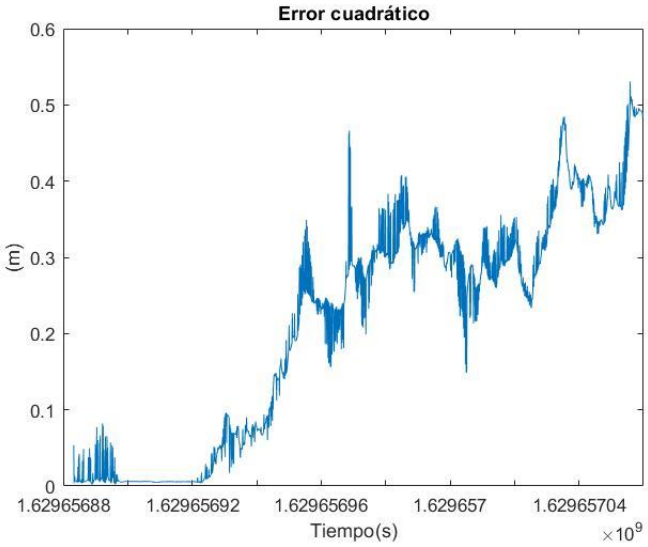


Figura 5.6 Error cuadrático con trayectoria guardada de MH01

Como se puede apreciar el error ha sido reducido en ciertos puntos de la trayectoria gracias a la información guardada, el algoritmo VINS-Mono mejora notablemente en ambientes ya conocidos, teniendo una mayor utilidad para entornos previamente visitados.

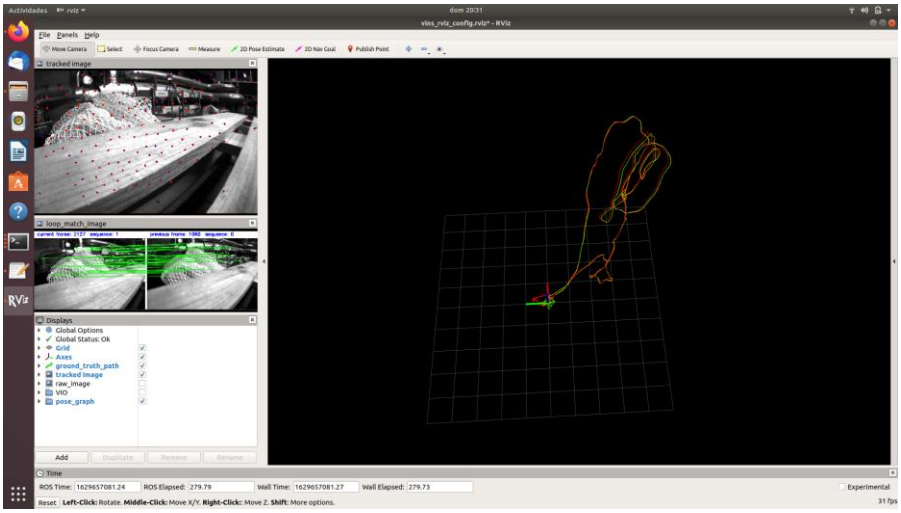


Figura 5.7 RVIZ con trayectoria guardada de MH01

En el visualizador RVIZ se puede observar en rojo la trayectoria ground truth, en verde la actual trayectoria estimada por VINS-Mono y por último en naranja la trayectoria guardada del primer experimento.

5.2 Configuración

Lo más importante en la configuración será indicar en el archivo de configuración euro.yaml los topics a los que debe atender el algoritmo, habrá un topic para la cámara y otro para el IMU.

Código 5.3 Archivo de configuración euroc.yaml

```
%YAML:1.0
#common parameters
imu_topic: "/raw_imu"
image_topic: "/camera/rgb/image_raw"
output_path: "/home/german"
#camera calibration
model_type: PINHOLE
camera_name: camera
image_width: 640
image_height: 480
distortion_parameters:
   k1: 0
   k2: 0
   p1: 0
   p2: 0
projection_parameters:
  fx: 381.3625
  fy: 381.3625
  cx: 320.5
  cy: 240.5
```

5.2.1 Configuración del IMU

En primer lugar, ejecutaremos rostopic list para ver los sensores que vienen por defecto en el quadrotor y saber si podemos utilizar alguno de ellos para nuestros experimentos.

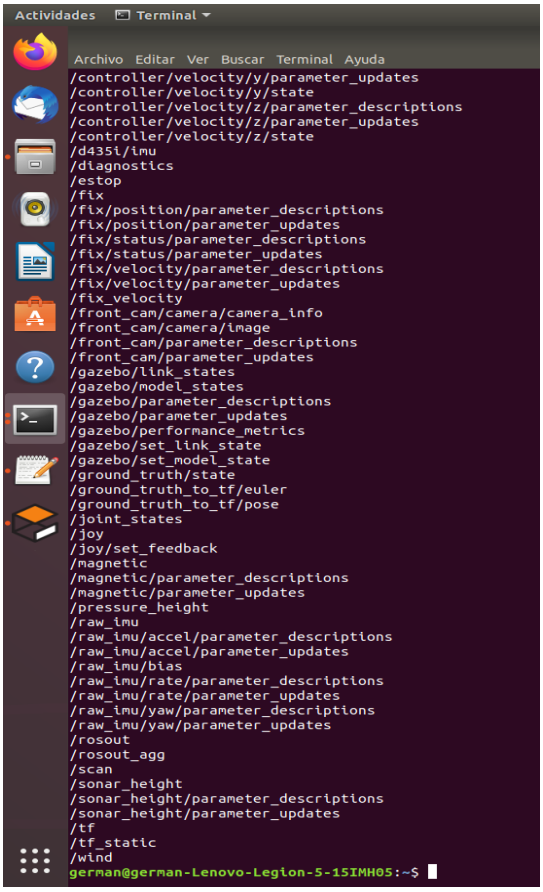


Figura 5.8 Ejecución de comando rostopic list

Encontramos que por defecto viene un sensor inercial que en un principio podremos usar, en un principio se trató de incluir un imu genérico mediante un plugin, pero la sincronización con la cámara no era buena y hubo que desechar esa opción. Así que finalmente se decidió usar el sensor por defecto que publica las medidas en el topic /raw_imu.

El mensaje que se requiere es de tipo sensor_msg/Imu que coincide con el contenido del topic /raw_imu comprobado gracias al comando de línea rostopic type /raw_imu

5.2.2 Configuración de la cámara

El paquete del dron viene con cámaras instaladas por defecto, pero ninguna de ellas nos servía ya que se mostraba una parte delantera del dron que no interesaba en la captura de características, y además el tamaño de la imagen hacía que no se capturasen suficientes características como para que el algoritmo funcionase correctamente. En concreto se trata de una cámara de 240x320 lo que al parecer no funcionaba de la manera óptima pues incluía una deriva mucho mayor que con la cámara real-sense.

Este es el resultado de ejecutar en la terminal rostopic echo /front_camera/info:

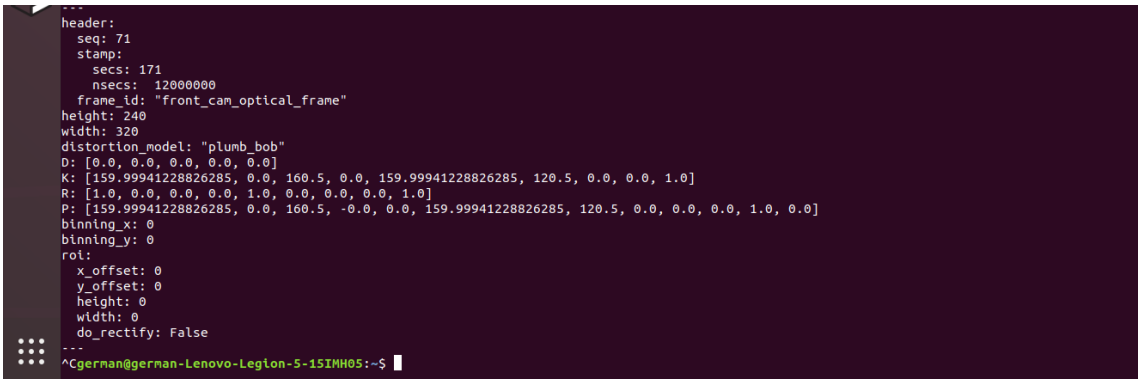


Figura 5.9 Parámetros cámara predeterminada

Por este motivo la decisión tomada fue incorporar manualmente en el URDF del paquete una cámara específica, la realsense-d435i que nos garantizaba un buen funcionamiento. El proceso de instalación de la cámara es muy sencillo, pero también es un paquete preparado para Ubuntu 16.04 por lo que habrá que modificar alguna parte de la instalación que viene en el tutorial. Además para el uso de la cámara habrá que tener otro paquete de sencilla instalación, el paquete ddynamic_reconfigure.

Código 5.4 Instalación de la cámara real-sense

```
//Instalación de librerias
sudo apt-get install librealsense2-dkms
sudo apt-get install librealsense2-utils
//Descarga del paquete e instalación
git clone https://github.com/IntelRealSense/realsense-ros.git
//Compilación del entorno de trabajo
catkin_make clean
   catkin_make -DCATKIN_ENABLE_TESTING=False -
   DCMAKE_BUILD_TYPE=Release
catkin_make install
//Configuración del entorno
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Una vez instalada si adjuntamos la cámara D435i se incluyen los topics necesarios para obtener información de sus cámaras.

Para unir la cámara con el dron modificamos el archivo urdf correspondiente, añadimos dos cámaras desplazadas una cierta distancia para comprobar con cuál el algoritmo es más eficiente, una será una cámara monocular y otra una cámara de profundidad. Aquí se muestran las modificaciones del código URDF para lanzar la simulación del hector_quadrotor junto con la cámara.

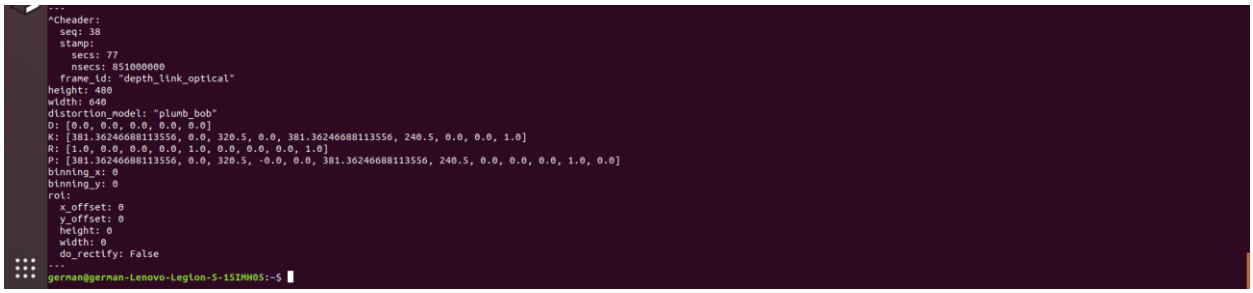


Figura 5.10 Parámetros cámara real-sense

Para unir la cámara que usaremos con el dron añadimos un brazo que uniremos al cuerpo del quadrotor así como un gimbal que irá unido a la cámara.

Dentro del urdf incluimos la ruta al archivo d435.dae que incluimos junto con los demás meshes del hector_quadrotor, también es importante incluir el plugin “kinect_camera_controller” ("libgazebo_ros_openni_kinect.so") en el que configuramos la frecuencia de captación de imágenes, los nombres de los topics en los que queremos publicar la información de la cámara así como los parámetros de distorsión.

Un archivo .dae (Digital Asset Exchange) es un archivo en formato XML COLLADA que se utiliza para mostrar gráficos.

El archivo .dae que necesitamos copiar y trasladar a nuestro paquete del dron es fácilmente localizable en el paquete realsense dentro de la carpeta realsense2_description/meshes.

A continuación parte del código perteneciente al archivo quadrotor_base.urdf.xacro que usaremos en la simulación:

Código 5.5 Parte de las modificaciones en quadrotor_base.urdf.xacro

```
<joint name="revolute_joint" type="fixed">
 <origin
    xyz="0.0 -0.0 0.0"
    rpy="0 0 0" />
    <parent link="base_link" />
    <child link="revolute_link" />
    <axis xyz="0 0 1" />
    <dynamics damping="0.01" friction="0.0" />
 </joint>
……..
<link name="camera_link">
      <visual>
        <origin xyz="-0.0 -0.0 0.0 " rpy="0 0 0" />
        <geometry>
          <mesh
filename="package://hector_quadrotor_description/meshes/quadrotor/d435.dae" />
        </geometry>
        <material name="white">
          <color rgba="0 0 0 1"/>
        </material>
      </visual>
….
<plugin name="kinect_camera_controller"
filename="libgazebo_ros_openni_kinect.so">
<cameraName>camera</cameraName>
      <alwaysOn>true</alwaysOn>
      <updateRate>40</updateRate>
      <imageTopicName>rgb/image_raw</imageTopicName>
```

5.3 Configuración del .launch

El propio paquete de hector_quadrotor nos ofrece dos demos para lanzar y simular en gazebo, una de ellas en exterior y otra de ellas en interior. Pero en lo que a este trabajo respecta ninguna de los dos nos era útil por lo que se modificó el archivo hector_quadrotor_demo/launch/outdoor_flight.launch para que en vez de aparecer en un entorno montañoso con pocas características apareciese en el mundo que nosotros quisiésemos configurar.

Se configuró para que el dron apareciese en un mundo vacío y gracias a los modelos que nos ofrecen las librerías de gazebo íbamos incluyendo edificios, personas, coches, etc.

Es importante tener en cuenta que si se incluyen montañas, promontorios u otros elementos a gran distancia y de gran tamaño confunden al algoritmo ya que detecta muchos puntos no singulares en la lejanía.

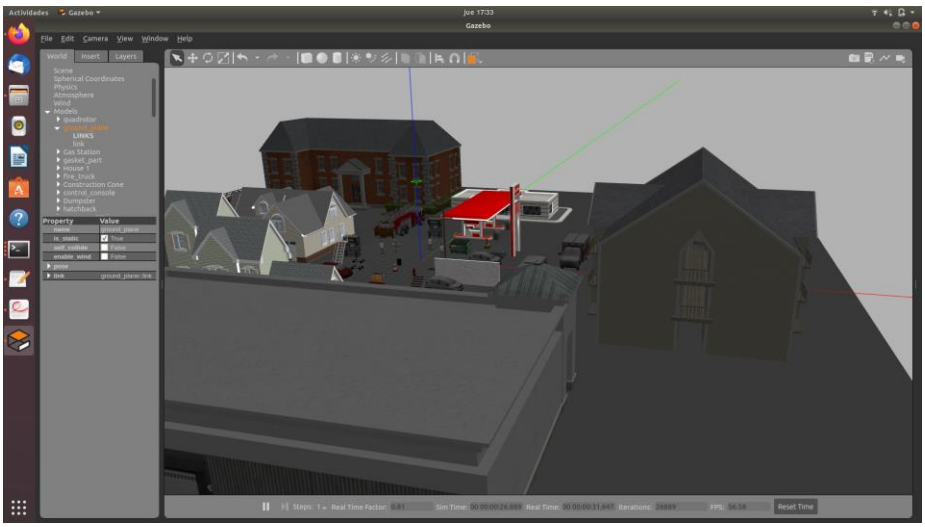


Figura 5.11 Mundo ejemplo en Gazebo

5.4 Calibración de parámetros extrínsecos

La calibración de la cámara con respecto al IMU es compleja, los parámetros internos son realmente fáciles de conseguir accediendo al topic donde se publica la información correspondiente, pero para la calibración externa no es posible obtener el sistema de coordenadas del IMU

La calibración de la cámara con respecto al IMU es compleja, los parámetros internos son realmente fáciles de conseguir accediendo al topic donde se publica la información correspondiente, pero para la calibración externa no es posible obtener el sistema de coordenadas del IMU, pero este problema es solucionado internamente por el algoritmo VINS-Mono ya que accediendo de nuevo al archivo de configuración y asegurándose de que el parámetro estimate_extrinsic esté configurado a 2, el algoritmo solo nos calculará los parámetros extrínsecos. Si se tiene una información completa y fiable de los parámetros extrínsecos el parámetro debe ser puesto a 0, y si se conocen, pero no se tiene una alta seguridad en la veracidad de los mismos, debe ponerse a 1. La diferencia entre ponerlo a 0 o a 1 es mínima y casi inapreciable ya que la calibración online que realiza el algoritmo es

verdaderamente rápida.

Los resultados de la calibración extrínseca los vuelca en un fichero .csv llamado extrinsic_parameter.

Para ello solo tenemos que crear un entorno rico en puntos característicos, es decir, con muchos objetos para que el estimador pueda trabajar correctamente y rotar la cámara a una cierta velocidad.

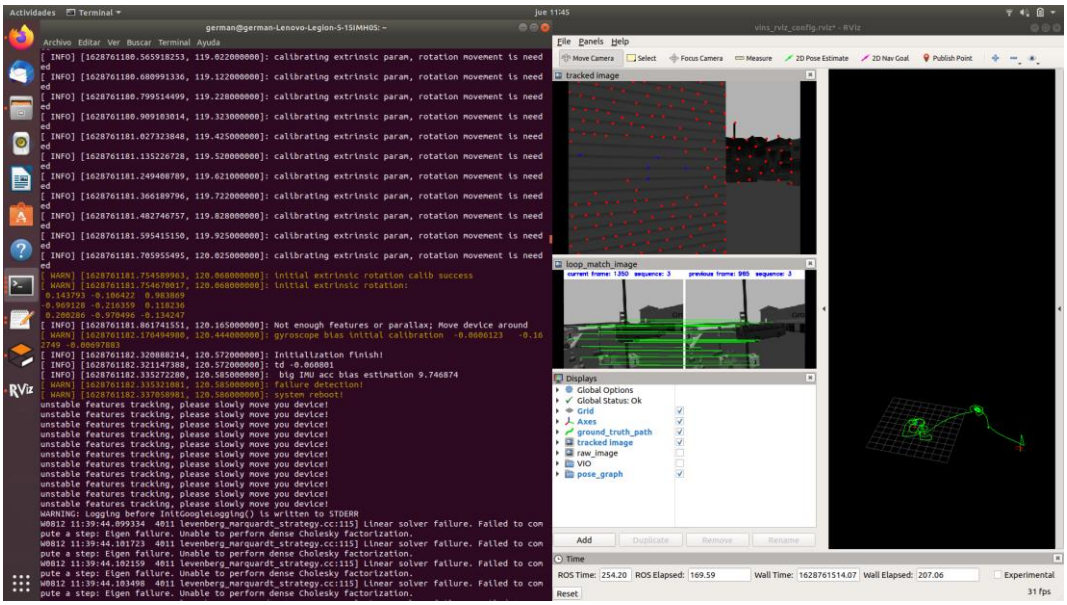


Figura 5.12 Calibración parámetros extrínsecos

Como se puede observar en el RVIZ en la parte derecha de la imagen la línea verde se corresponde con la odometría visuo-inercial del algoritmo, y es visible que se ha ido rotando hasta conseguir unos parámetros extrínsecos, una matriz de rotación entre los ejes de la cámara y del IMU y un vector de traslación.

6 RESULTADOS EXPERIMENTALES

En este apartado del trabajo vamos a centrarnos en comprobar la fiabilidad del algoritmo ante diferentes situaciones, en tratar de realizar una fusión sensorial de la odometría del VINS-Mono con un sensor GPS y, finalmente realizar un pequeño archivo de control para despegue del dron y vuelo horizontal realimentado en todo momento por la odometría visuo-inercial del algoritmo.

6.1 Simulaciones con el algoritmo

Para realizar estas simulaciones nos hemos servido de un nodo de teleoperación que nos permitirá controlar el movimiento del dron mediante teclado.

Este nodo estaba ya incluido en hector_quadrotor y es muy común en este tipo de paquetes.

Para lanzar la simulación se deberán seguir estos pasos:

Código 6.1 Pasos para ejecutar el algoritmo VINS-Mono en Hector_quadrotor

```
//Se lanza la simulación con el mundo vacío, posteriormente se
//le incluirán objetos
roslaunch hector_quadrotor_demo outdoor_flight_gazebo.launch
//Activación de motores
rosservice call /enable_motors "enable: true"
//Ejecución del nodo de teleoperación
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
//Guardamos en bolsas los datos que nos interesan para su
//posterior interpretación
rosbag record -O vins /vins_estimator/odometry
rosbag record -O gt /ground_truth/state
//Se lanza el estimador en sí y el visualizador
roslaunch vins_estimator euroc.launch
roslaunch vins_estimator vins_rviz.launch
```

6.1.1 Experimento sin calibrar

En un principio se ha probado el algoritmo sin calibrar, y según lo esperado los resultados son nefastos. Esto visibiliza la importancia de una buena calibración.

En el caso de este algoritmo se nos proporciona una calibración online en tiempo real, de no ser así habría que recurrir a una calibración en gazebo o un método de calibración como Kalibr.

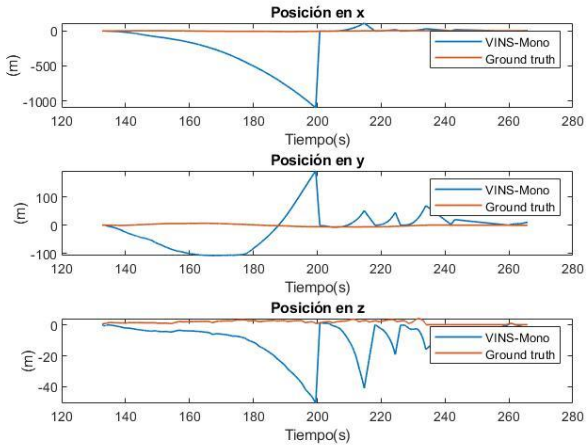


Figura 6.1 Evolución temporal experimento sin calibrar

Se enfrentan en esta gráfica la odometría ground truth que es una odometría que nos proporciona la simulación y que se supone perfecta respecto a en azul la odometría de VINS-Mono.

Es apreciable la deriva de hasta 1000 metros en la coordenada x, aun así, cómo hemos visto en el marco teórico el algoritmo tiene una capacidad de recuperación y de volver a un punto correcto.

6.1.2 Experimento calibrado

El siguiente experimento es tras incluir la matriz de rotación y el vector de traslación en nuestro archivo de configuración euroc.yaml

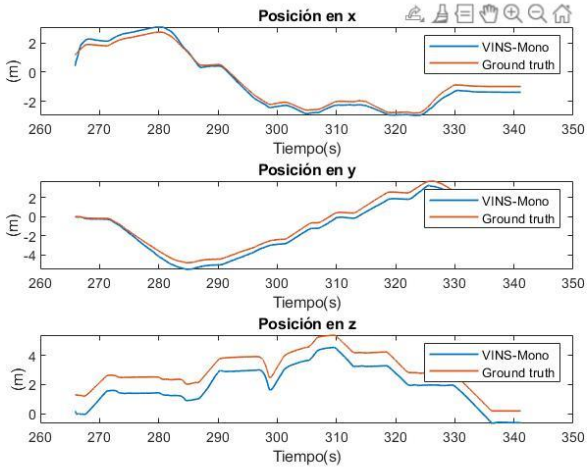


Figura 6.2 Evolución temporal de experimento calibrado

Como observamos, la calibración es un paso fundamental en el algoritmo llegando a reducir los errores de la odometría en unas magnitudes realmente considerables en relación con el tamaño de un dron.

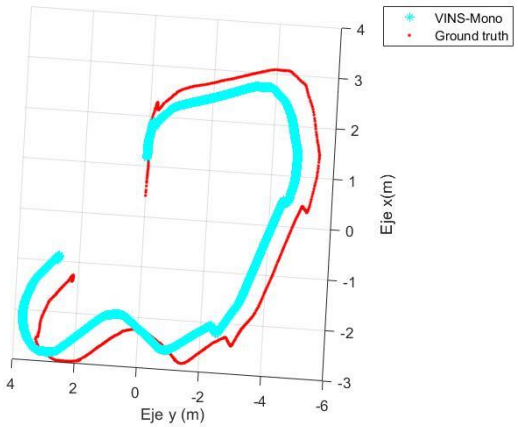


Figura 6.3 Visualización 3D del vuelo, experimento calibrado

Como se puede observar en las gráficas los resultados son bastante afines a la realidad.

Para medir el error estimado se calcula el error cuadrático dando un resultado como este:

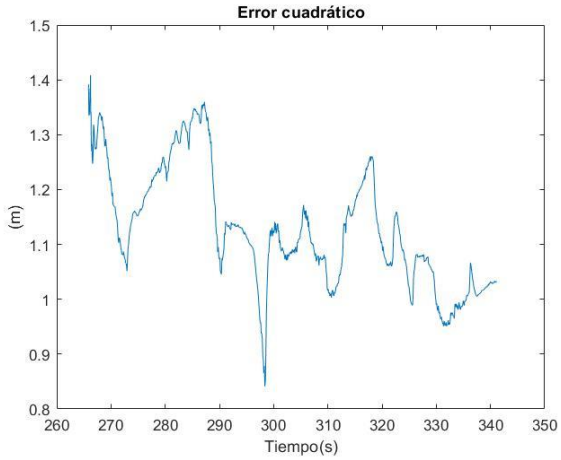


Figura 6.4 Error cuadrático experimento calibrado

El error es aceptable considerando las dimensiones habituales de un quadrotor.

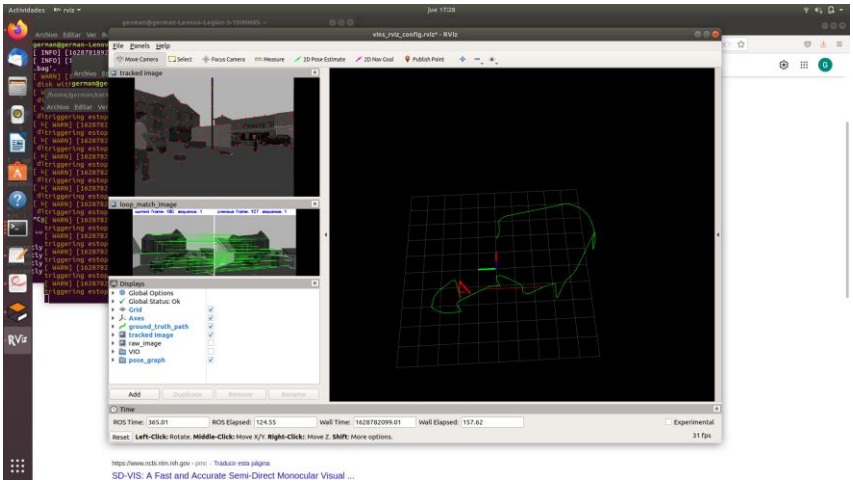


Figura 6.5 RVIZ final experimento calibrado

En la siguiente imagen podemos observar tres detalles importantes gracias al visualizador tridimensional RVIZ, la primera de ellas es la detección de puntos característicos.

Como se puede observar en la esquina superior izquierda del RVIZ, los puntos rojos son puntos característicos detectados por el algoritmo, los puntos candidatos a ser característicos saldrían en azul, pero en el momento de la toma de la captura de pantalla el quadrotor llevaba un tiempo en la misma posición por lo que había tenido tiempo suficiente para procesar y distinguir entre puntos característicos y no.

La segunda, justo debajo de la detección de puntos característicos, es la correspondencia entre imágenes, estableciendo las correspondencias de las dos imágenes y consiguiendo así uno de los pasos más importantes para la estimación de posición.

En el resto de la pantalla de RVIZ, se muestra frente a un fondo negro la estimación de la posición del dron durante el recorrido realizado, mostrando además la pose actual del robot mediante una pirámide roja que apunta en todo momento hacia donde esté apuntando el dron.

Usando la herramienta rqt_graph podemos comprobar mediante un esquemático los nodos activos durante la simulación y como se comunican entre ellos para aclararnos un poco las ideas acerca del funcionamiento del experimento:

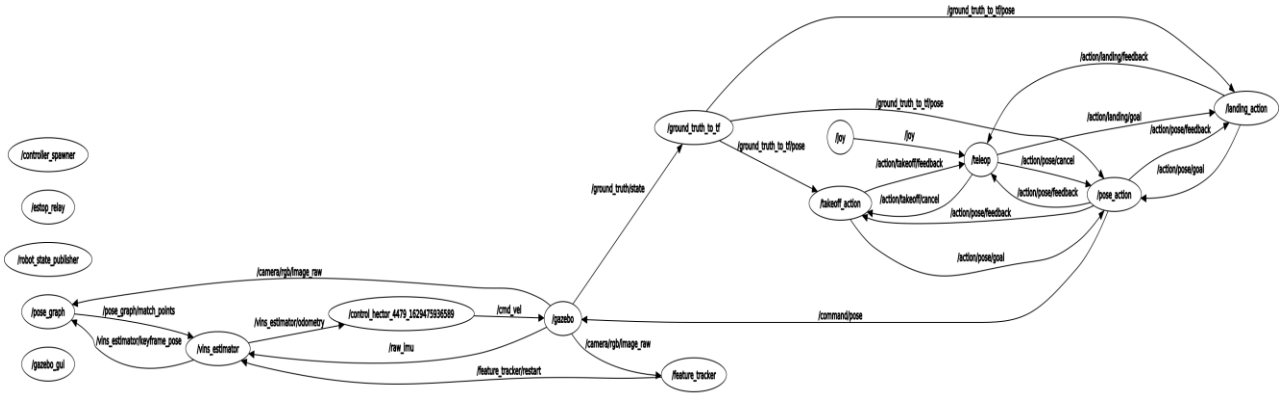


Figura 6.6 RQT_Graph del experimento 6.1.2

6.2 Fusión sensorial con GPS

La fusión sensorial es la agrupación de información de varios sensores con intención de mejorar de alguna manera las medidas que darían esos mismos sensores de manera individual y aislada.

En el paquete original de los autores no viene contemplada la fusión sensorial con elementos externos, aplicando ellos ya una fusión sensorial interna entre el IMU y la odometría visual.

Combinar la información de la odometría visuo-inercial junto con el GPS es una idea muy buena ya que hay situaciones en las que no hay disponibilidad de usar el GPS, ya sea por falta de conexión, perdidas de información en el camino, entornos en los que no funcione, etc.

Todos los algoritmos de estimación de la posición poseen una cierta deriva, por ese motivo mediante la fusión sensorial se trata de corregir la misma.

Para realizar la fusión en ROS emplearemos un conocido paquete llamado robot_localization, que además de permitirnos realizar la fusión sensorial, también nos permitirá traducir las coordenadas del GPS de latitud, altitud y longitud en coordenadas expresadas en cartesianas.

El primer paso será incluir el GPS mediante un plugin para que publique en un topic llamado /fix, el nombre del topic es configurable, así como la frecuencia de actualización.

Código 6.2 Inclusión del plugin de gps

```
<plugin name="gazebo_gnss" filename="libhector_gazebo_ros_gps.so">
 <updateRate>2</updateRate>
 <bodyName>base_link</bodyName>
 <topicName>fix</topicName>
 <velocityTopicName>vel</velocityTopicName>
 <referenceLatitude>REMOVED</referenceLatitude>
 <referenceLongitude>REMOVED</referenceLongitude>
 <referenceAltitude>REMOVED</referenceAltitude>
 <status>2</status>
 <service>8</service>
</plugin>
```

Una vez configurado el GPS deberemos tratar de conseguir cambiar el tipo de mensaje que publica que es de tipo sensor_msg/NavSatFix, mientras que nosotros para fusionarlo con el paquete robot_localization necesitamos mensajes de odometría.

Dentro del paquete descargado e instalado correctamente existe la posibilidad de correr un nodo llamado navsat_transform_node que se encarga de preparar los datos provenientes del GPS para su posterior fusión.

Este nodo necesita un mensaje de tipo sensor_msgs/NavSatFix que tenemos al incluir el plugin del gps, un mensaje del IMU referenciado al mundo y un mensaje de odometría que contenga la posición actual del robot.

Se configura de la siguiente manera:

Código 6.3 Configuración del nodo navsat_transform

```
<launch>

 <node pkg="robot_localization"
type="navsat_transform_node"
name="navsat_transform_node" clear_params="true">

<rosparam command="load"
file="$(find robot_localization)/params/navsat_transform_template.yaml" />

    <param name="broadcast_utm_transform" value="true"/>
    <param name="publish_filtered_gps" value="true"/>

    <param name="magnetic_declination_radians" value="0"/>

    <param name="yaw_offset" value="0"/>


    <remap from="/raw_imu" to="/imu_topic"/>
    <remap from="/odometry/filtered" to="/ground_truth/state "/>
    <remap from="/fix" to="/gps_pose"/>


 </node>
</launch>
```

De una manera esquemática el uso del nodo navsat queda bien explicado de esta forma:

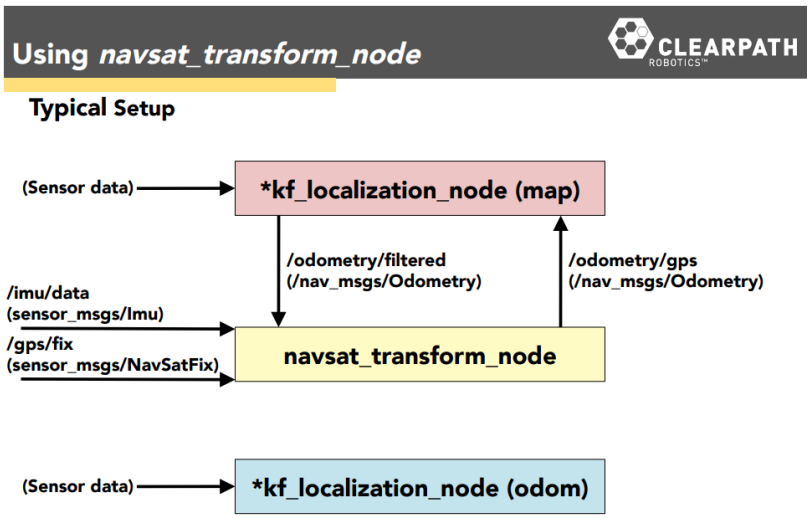


Figura 6.7 Configuración del nodo navsat_transform

De esta forma conseguimos publicar en gps_pose un mensaje del tipo nav_msgs/Odometry y así pasar a la fusión.

En este apartado se implementará un filtro extendido de Kalman o EKF, que es la forma del filtro de Kalman no lineal. El filtro de Kalman extendido no es lineal en sí, sino que se linealiza en torno a una primera estimación actual de la media y la covarianza.

Los filtros de Kalman siguen un método iterativo con dos etapas, la primera es la actualización de la medición y luego, realizan las predicciones.

Una vez escogido el filtro que vamos a poner a funcionar es necesario configurar un archivo de parámetros en el que indicaremos al nodo encargado de la fusión qué medidas queremos unir y cómo.

A continuación muestro los parámetros más relevantes en el archivo .yaml para su correcta configuración

Código 6.4 Configuración del archivo de parámetros para fusión

```
//Es importante poner el parámetro de dos dimensiones a falso ya que si no no usará
//medidas tridimensionales
  two_d_mode: false
…
//Por un lado fusionaremos la salida de odometría del estimador VINS-Mono
odom0: /vins_estimator/odometry
odom0_config: [true,  true,  true,
               true, true, true,
               true, true, true,
               false, false, false,
               false, false, false]
….
//GPS
odom1: /gps_pose
odom1_config: [true, true, true,
               false, false, false,
               false, false, false,
               false, false, false,
               false, false, false]
odom1_queue_size: 1
//Las matrices de covarianza del ruido también se configuran, son matrices diagonales
//de 14x14, pero en nuestro caso las dejaremos por defecto.
```

Las matrices booleanas expresan que partes de las medidas queremos fusionar, al igual que en un IMU lo normal sería fusionar medidas de aceleración, en el GPS, al convertir las medidas a coordenadas cartesianas, escogeremos las tres medidas de posición en x, y, z.

[x_pos, y_pos, z_pos,
roll, pitch, yaw,
x_vel, y_vel, z_vel,
roll_vel, pitch_vel, yaw_vel,
x_accel, y_accel, z_accel]

Poniendo por ejemplo la primera posición a true estaremos incluyendo la posición en x para la posterior fusión. En el caso de la odometría del VINS-Mono al ser un quadrotor es recomendado en el paquete cubrir una gran parte de posiciones a true, excepto los de aceleración.

Una vez configurado el archivo de parámetros del filtro podemos proceder a probarlo en un entorno rico en características.

Para ello los comandos necesarios serán, junto con los realizados para el experimento 6.1 los siguientes:

Código 6.5 Lanzamiento del nodo navsat_transform junto con el nodo para fusión sensorial

```
cd catkin_ws/src/robot_localization-melodic-devel/launch
roslaunch ekf_template.launch
cd catkin_ws/src/robot_localization-melodic-devel/launch
roslaunch navsat_transform_template.launch
rosbag record -O odom_fl /odometry/filtered
```

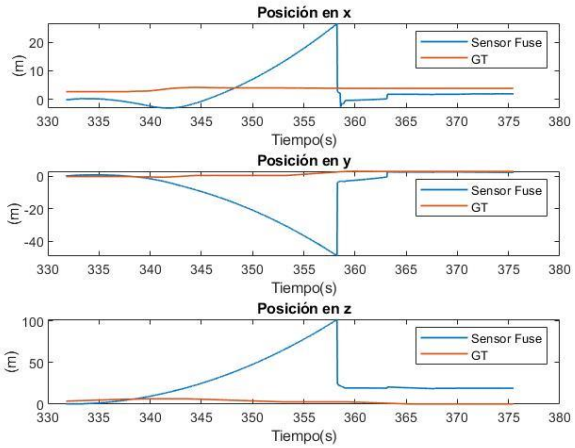


Figura 6.8 Evolución de la fusión sensorial frente a ground truth

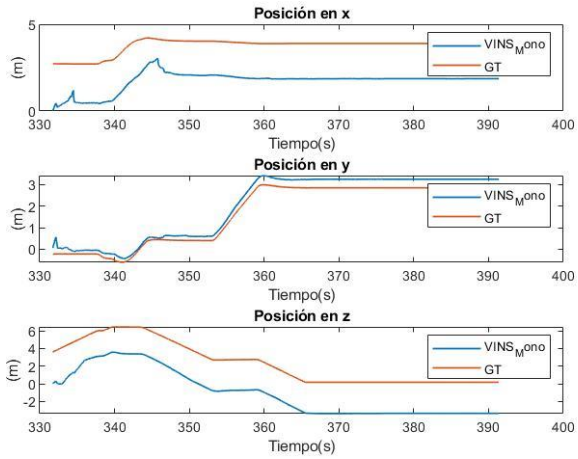


Figura 6.9 Evolución de VINS-Mono en experimento de fusión sensorial

Es apreciable a simple vista que la fusión sensorial presenta un peor comportamiento a pesar de haber intentado ajustar la frecuencia de publicación del GPS y haber limitado el número de muestras obtenidas para fusión a sabiendas que el GPS suele dar medidas a una frecuencia aproximada de 2 Hz.

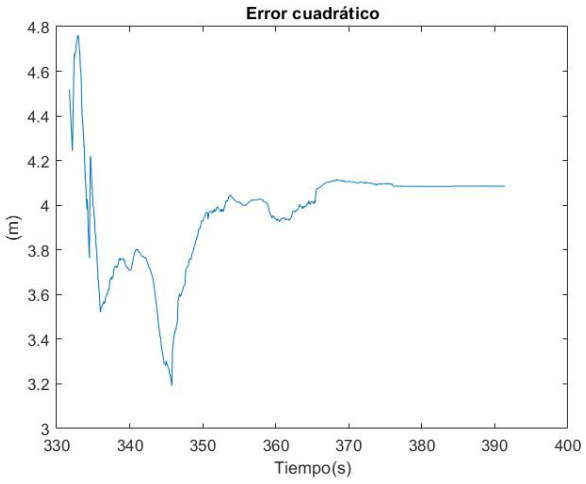


Figura 6.10 Error cuadrático en fusión sensorial

Los resultados obtenidos no son los esperados, ya que se incluía la fusión sensorial con el objetivo de mejorar la odometría pero en cambio la ha empeorado. Se han probado el cambio de parámetros y muchas más opciones, pero no mejoraban cuantitativamente los resultados.

Se desconoce si es un error de implementación.

El entorno en el que se desarrolla este experimento es uno creado en Gazebo a partir de la base de modelos proporcionadas por el simulador

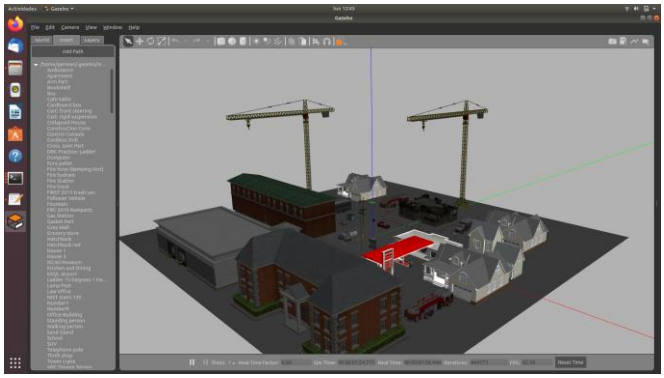


Figura 6.11 Entorno creado para experimento 6.2

6.3 Control de vuelo con realimentación de odometría visuo-inercial

En este apartado se desarrolla un control proporcional para el despegue del UAV hasta una cierta altura introducida por el usuario mediante teclado, una vez alcanzada la altura el UAV deberá desplazarse horizontalmente para comprobar si la odometría realizada por VINS-Mono es eficaz en esta tarea.

Todo ello se realiza mediante un script escrito en python llamado control.py

La odometría realizada por el algoritmo no es inmediata, por ello hay que mover el dron manualmente antes de lanzar el nodo de control.

Se proporciona un pseudo-código del nodo de control:

Código 6.6 Pseudo-código del archivo de control control.py

```
-Inclusión de las librerías apropiadas para el código y para los mensajes usados

-Inicializar variables de posición

-función subscriptora al topic de odometría de VINS-Mono
        Actualizar variables de posición

-función principal
    Inicializar constantes de proporcionalidad
    Iniciar nodo de control
    Establecer frecuencia de actualización
    Declarar subscriptor y publicador
    Pedir coordenada z por pantalla
    Almacenarla
    Mientras que distancia_actual_z-distancia_referencia_z menor que 0.8
            Calcular comando para mandar por /cmd_vel
            Actualizar incremento de distancia en z
            Publicar
    Fin Mientras

  Inicializar referencia en x
    Mientras que distancia_actual_x-distancia_referencia_x menor que 0.8
            Calcular comando para mandar por /cmd_vel
            Actualizar incremento de distancia en x
            Publicar
    Fin Mientras
```

Los resultados son los que siguen:

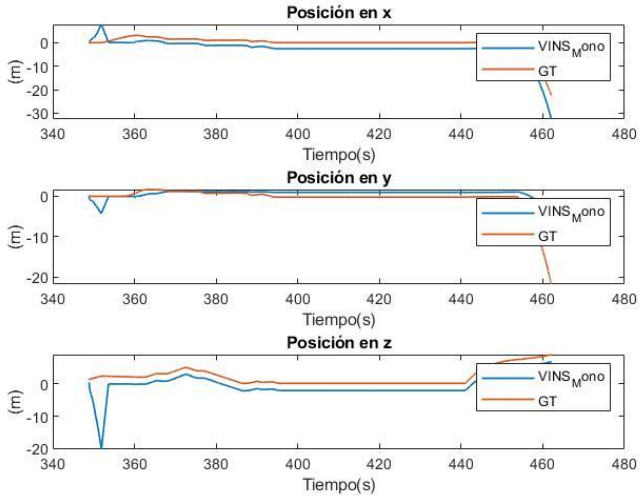


Figura 6.12 Evolución de la VIO con control

Los picos iniciales se deben a una caída del robot cuando el algoritmo VINS-Mono estaba comenzando a funcionar, se chocó con un objeto, pero no es importante para el experimento, el UAV alcanza la posición de referencia en el eje z con solvencia.

Se puede observar el pico en posición z hasta una altura de cinco metros que fue lo que se indicó en este experimento por teclado y después recorre horizontalmente una cierta distancia hasta que se finaliza el nodo de control rondando los 450 segundos.

Los resultados son aparentemente buenos a excepción de los picos iniciales y el momento de la terminación del nodo de control.

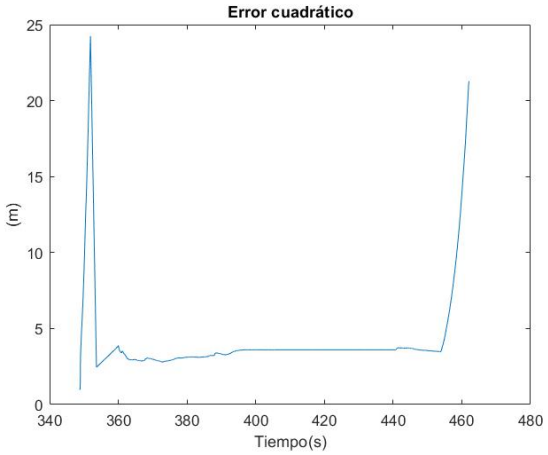


Figura 6.13 Error cuadrático en experimento de control del UAV

Los errores son aceptables teniendo en cuenta la deriva implícita del algoritmo.

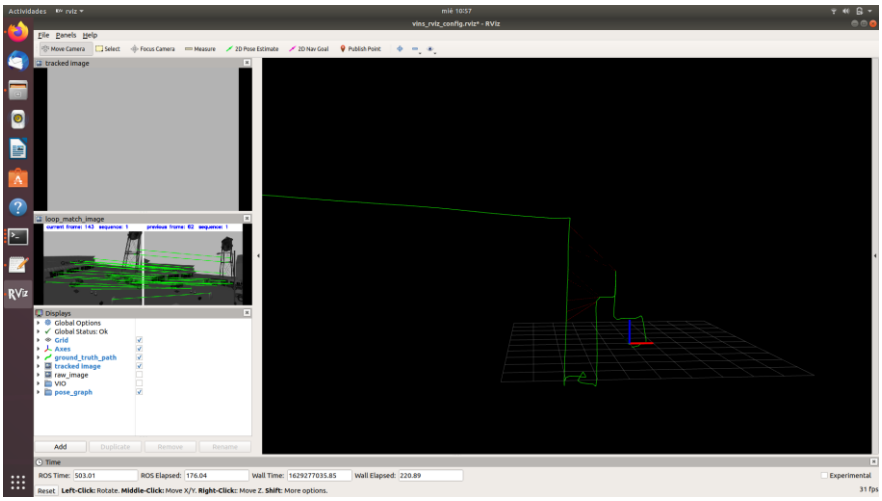


Figura 6.14 RVIZ en el experimento de control

Lo comentado anteriormente se puede verificar en la creación de la trayectoria en RVIZ, inicialmente se mueve mediante el nodo de teleoperación y cuando el algoritmo comienza a estimar la posición podemos arrancar el nodo de control.

En definitiva, podemos concluir que controlando el dron a una velocidad adecuada el algoritmo funciona correctamente, pero se hicieron otras pruebas con un control más agresivo en el que el dron alcanzaba velocidades superiores y el algoritmo no respondió de la manera esperada.

6.4 Control de vuelo con reutilización del grafo de poses

VINS-Mono posee una gran ventaja y es que puedes guardar una trayectoria ya descrita por tu dron para si después se realiza alguna parecida poder detector similitudes y mejorar la precisión del algoritmo considerablemente.

Para llevar a cabo el almacenamiento de los fotogramas clave y demás información que utilizará en futuros movimientos el algoritmo, es necesario poner en el archivo de configuración un directorio donde guardarlo, esto es fácilmente realizable cambiando en el archivo de configuración ubicado en

(TU_DIR_VINS/config/euroc/euroc.yaml) el path de pose_graph_save_path al lugar donde desees guardar dicha información.

Una vez realizado el primer experimento para el que se desea guardar el grafo de poses, es necesario en la terminal donde se ejecuta el nodo estimador teclear una ‘s’ y pulsar enter. De esta manera se guardará toda la información requerida.

Una vez realizado este sencillo paso en el mismo archivo de configuración ponemos el parámetro load_previous_pose_graph a 1 siempre antes de realizar el próximo experimento

En nuestro caso realizamos el control de vuelo en un escenario diferente al del experimento 6.3 pero parecido ya que ese mundo no fue guardado para este experimento.

A continuación, muestro los resultados para el primer vuelo, en el que no se utilizaba esta poderosa herramienta de VINS-Mono:

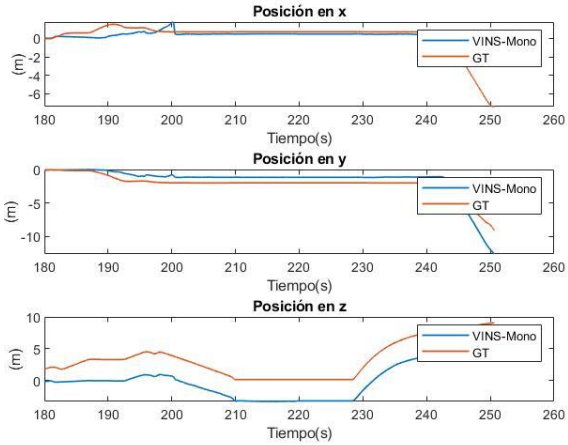


Figura 6.15 Evolución control sin reutilización del grafo de poses

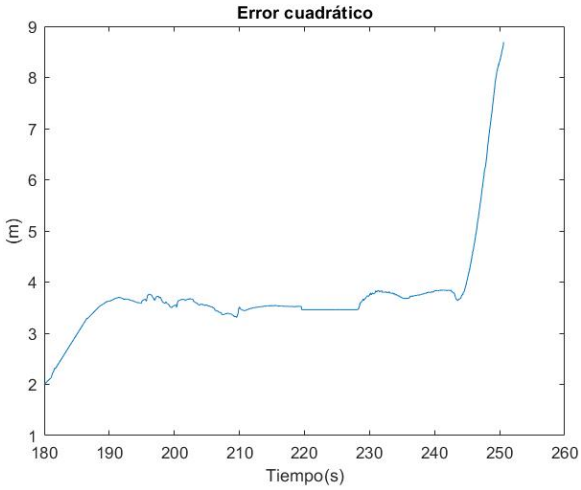


Figura 6.16 Error cuadrático sin reutilización del grafo de poses

Como veremos a continuación los resultados mejoran notablemente cargando una trayectoria realizada con anterioridad.

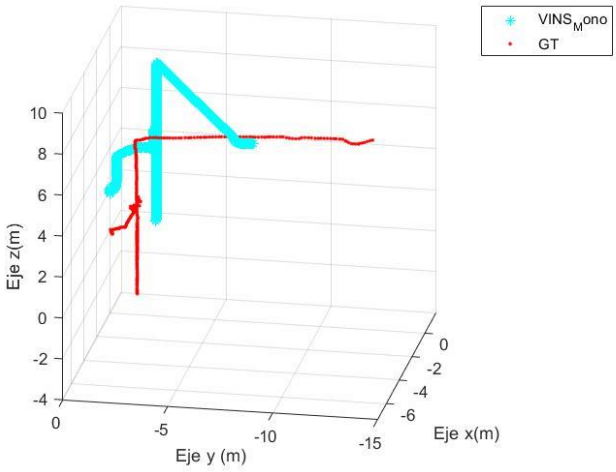


Figura 6.17 Trayectoria tridimensional sin reutilización del grafo de poses

En la siguiente figura se pueden observar los resultados para el segundo recorrido, no será exactamente el mismo que el anterior ya que para que el nodo de estimación de la posición comience a funcionar correctamente es necesario moverlo previamente, a pesar de ello se ha conseguido que las trayectorias sean realmente similares, intentando empezar el control en el mismo punto de partida que en el primer experimento.

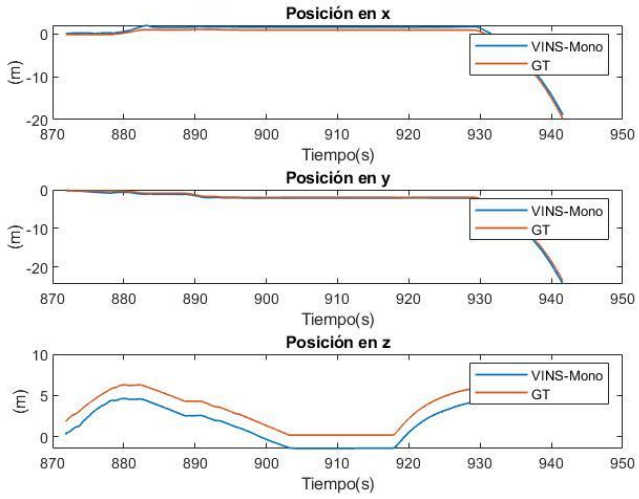


Figura 6.18 Evolución con reutilización del grafo de poses

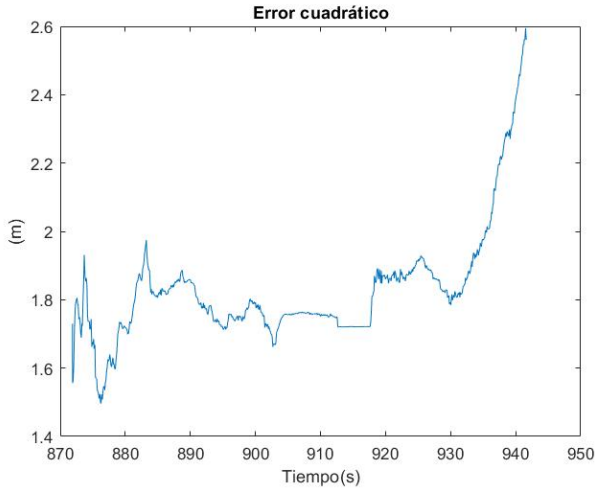


Figura 6.19 Error cuadrático con reutilización del grafo de poses

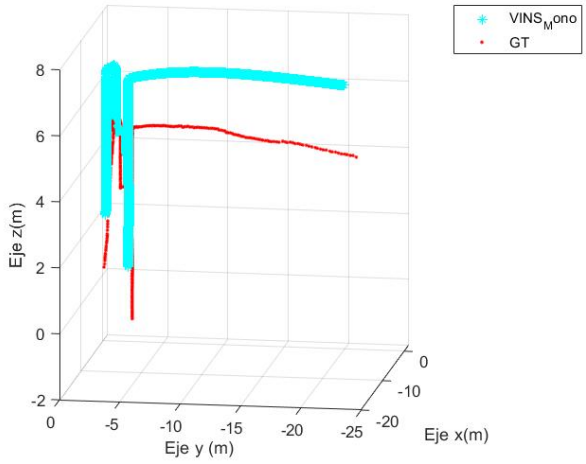


Figura 6.20 Trayectoria tridimensional con reutilización del grafo de poses

Los resultados son notablemente mejores, siendo el error cuadrático prácticamente la mitad del que se obtuvo sin aplicar esta herramienta, en conclusión VINS-Mono trabaja mucho mejor en entornos ya conocidos de los que pueda extraer características y cotejarlas con datos previamente almacenados.

Esta opción es realmente útil si se va a operar en lugares similares repetidamente.

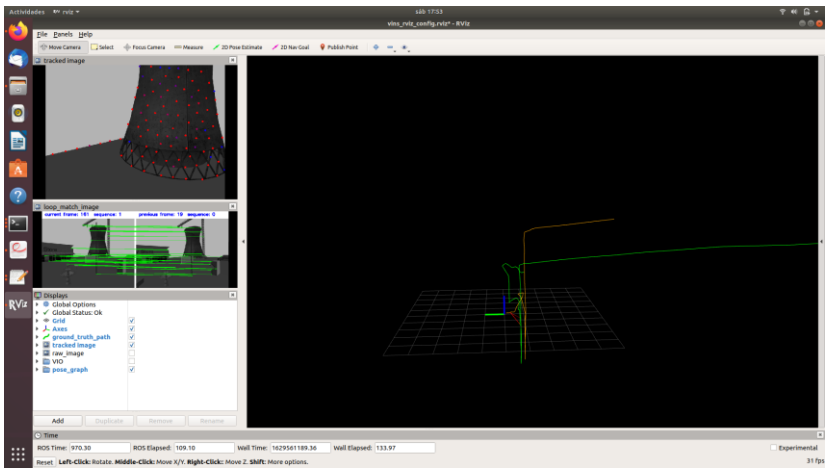


Figura 6.21 RVIZ de experimento de control con reutilización del grafo de poses

En la última figura se aprecia una trayectoria en amarillo que es la trayectoria guardada del primer vuelo de la aeronave y la trayectoria verde que representa la actual.

6.5 Control de vuelo PID con detección de obstáculos

En esta ocasión se implementa un control discreto PID, proporcional, integral y derivativo, desacoplado en el topic /cmd_vel para x, y, z y para yaw. Este control se realiza con el fin de mejorar el seguimiento de waypoints.

Los waypoints serán indicados por el usuario mediante teclado, se ha establecido como predeterminado que se escojan tres puntos en el sistema de coordenadas cartesianos.

El código de control se suscribe a /ground_truth/state que contiene un mensaje de tipo Odometry, extraemos las posiciones actuales, así como el roll, pitch y yaw.

Posteriormente en la función principal con estos valores y teniendo en cuenta los previos se procede a calcular la actuación necesaria para cada grado de libertad, en la sección de códigos se muestra el código completo.

Para la detección de obstáculos se ha aprovechado el topic /scan que contiene un mensaje de tipo LaserScan.

Este mensaje contiene varios apartados, nuestro programa se suscribirá al topic y utilizará varios de ellos.

En primer lugar, ya que no se tenía acceso directo a los parámetros del láser, rango en ángulo, número de rayos emitidos, etc., se decidió implementarlo mediante código, se extrae la longitud de las medidas de distancia del láser y se realiza un bucle en el que gracias al ángulo mínimo provisto por el topic y al incremento de ángulo se recorren todos los rayos, se calcula el ángulo y se recoge la distancia.

Una vez obtenido ambos parámetros se procede a, mediante un umbral estipulado, si el ángulo del rayo está comprendido entre -45 y 45 grados y si la distancia al objeto detectado es menor de un metro, se activa la condición de objeto detectado, que será usada por el programa para volver al último waypoint seguro.

Para la vuelta al último waypoint se utiliza un servicio ya implementado por la librería de hector_quadrotor en el que podemos suministrarle un waypoint para que acuda a él. No se ha utilizado este servicio en otros experimentos ya que el seguimiento de la trayectoria es extremadamente brusco y el propio algoritmo VINS-Mono no estimaba con fiabilidad la posición de ser usado.

Tras volver al último waypoint se le indica al usuario que se ha escogida una ruta insegura para el dron y que deberá introducir otra trayectoria.

Lamentablemente no ha sido posible probar el algoritmo realimentado con la odometría de VINS-Mono, ya que intentando mejorar la precisión del estimador las medidas que proporciona no son las adecuadas para que funcione establemente, probablemente por un problema con la versión del Eigen ya que no se computa correctamente la factorización Cholesky, haciendo erróneamente el cálculo matricial y afectando de esta manera aumentando la deriva, haciendo inestable el algoritmo.

Se ha intentado instalar desde las fuentes diferentes versiones de Eigen así como instalar desde el terminal pero ninguna versión ha conseguido que el algoritmo funcione como lo hacía en experimentos anteriores, siendo la configuración del algoritmo exactamente la misma que cuando si funcionó en experimentos previos.

Para comprobar el correcto funcionamiento del código de control se ha simulado en un principio el dron en un mundo en Gazebo sin obstáculos y se le han asignado tres waypoints arbitrarios, [0,0,3], [2,2,3] y [4,5,3]. Todas las distancias están expresadas en metros.

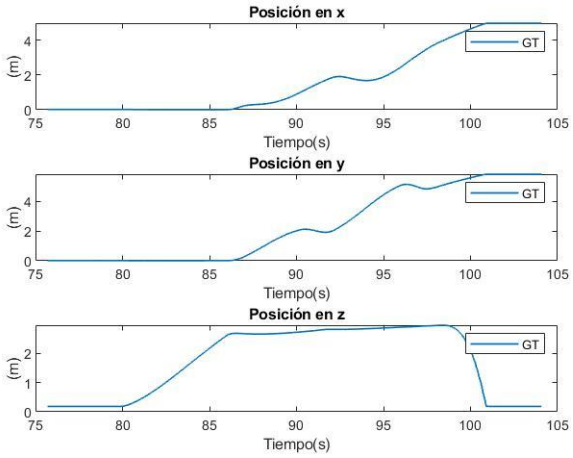


Figura 6.22 Evolución control PID

Cómo se puede observar el control tiene errores en régimen permanente pequeños, las constantes derivativas, integrales y proporcionales han sido determinadas por prueba y error.

Para el experimento en el que se pone a prueba la detección de obstáculos se introduce una casa en el mundo que interfiera en la trayectoria que deberá seguir el dron, la cuál es: [0,0,2], [2,0,2], [6,0,2].

El resultado es el que sigue:

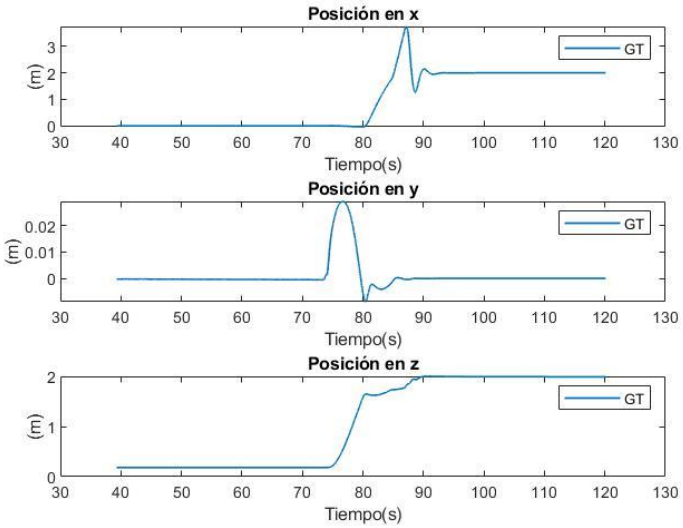


Figura 6.23 Evolución PID con obstáculo

Una vez se detecta el obstáculo aproximadamente en el segundo 86 aproximadamente se puede comprobar que el vehículo aéreo vuelve al waypoint número dos.

El entorno en el que se ha realizado esta prueba es realmente básico y se muestra a continuación:

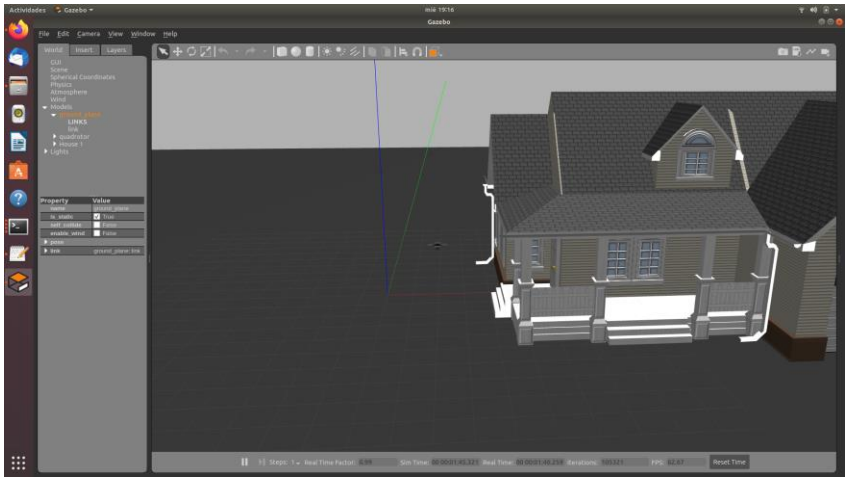


Figura 6.24 Entorno PID con detección de obstáculos

6.6 Código de MATLAB para interpretación de los experimentos

En esta sección se detalla una parte del código de MATLAB empleado para analizar el interior de los archivos .bag que se han ido recopilando durante el experimento:

Los datos de la odometría recogida en el topic /ground_truth/state se publican en el topic a una frecuencia muy superior a la frecuencia de publicación de la odometría visuo-inercial, para calcular el error debemos escalar los datos para que concuerden en el tiempo y que haya la misma cantidad de muestras.

Código 6.7 Código de MATLAB para representación en gráficas

```
bag1=rosbag('odom_2sf.bag')

bag2=rosbag('gt_2sf.bag')

//En el topic de ground truth se publica mucho antes que en el de la odometría
//del VINS-Mono, es necesario escalarlo

start1=bag1.StartTime;

end1=bag1.EndTime;

bag2select=select(bag2,'Time',[start1 end1])

//Se extraen de las bolsas el contenido del mensaje que requerimos
//junto con el tiempo de simulación
vins_x=timeseries(bag1,'Pose.Pose.Position.X');

t_gtx=timeseries(bag2select,'Pose.Pose.Position.X');


//Se aísla por un lado el tiempo y por otro los datos de posición

xvins=vins_x.Data;

tvins=vins_x.Time;



//Se muestra la evolución en posición respecto al tiempo
figure(1)
subplot(3,1,1);
plot(vins_x, 'LineWidth', 1);
hold on;
plot(t_gtx, 'LineWidth', 1);
title("Posición en x");

figure(2)
plot3(xvins,yvins,zvins,'r.');
grid;
```

Código 6.7 Código de MATLAB para representación en gráficas

```
s=length(tvins);

r=length(tgt);
//Interpolo para obtener el mismo número de muestras y así poder calcular
//el error mediante la resta en valor absoluto

for i=1:s

   for x=1:r

      if (abs(tvins(i)-tgt(x))<0.09)

         gt_x1(i,1)=gt_x(x,1);
         gt_y1(i,1)=gt_y(x,1);
         gt_z1(i,1)=gt_z(x,1);

      end

   end

end


//Por último se muestra el error cuadrático junto con el tiempo


error_x=abs(xvins-gt_x1);
error_y=abs(yvins-gt_y1);
error_z=abs(zvins-gt_z1);

error_2=sqrt(error_x.^2+error_y.^2+error_z.^2);
figure(3);
plot(tvins,error_2);
```

-

7 CONCLUSIONES

La odometría visuo-inercial es uno de los algoritmos en boga en el ámbito de la estimación de la posición, la localización de los robots es uno de los pilares fundamentales para la navegación autónoma. Por este motivo se investiga continuamente acerca de este tema.

El algoritmo VINS-MONO posee una cualidad muy ventajosa y es que se puede utilizar en cualquier situación, interior y exterior, ya que solo requiere de una cámara y un IMU.

Su uso de sensores es mínimo lo que puede permitir una autonomía mucho mayor en robots móviles terrestres y en voladores.

El algoritmo de relocalización permite corregir derivas como se ha visto en los experimentos realizados en distintas situaciones.

Como desventaja principal se puede comentar la dificultad de operar en entornos con visibilidad reducida, niebla, polvo, lluvia.

VINS-Mono es un algoritmo de estimación robusto y fiable cuando el movimiento del dron es controlado ya que se ha comprobado que ante movimientos bruscos las medidas pierden validez.

En el presente trabajo se ha incluido la fusión sensorial con GPS, una mejora que de ser realizada con éxito permitiría realizar estimaciones de alta eficacia y fiabilidad en multitud de entornos ya que combinaría dos de las odometrías más usadas en la actualidad.

Se ha probado el algoritmo en diferentes entornos, así como realizando un pequeño control proporcional en despegue y posterior vuelo horizontal, comprobando que realimentando el control con la odometría visuo-inercial funciona perfectamente, siempre que el control no sea brusco.

En definitiva, se ha comprobado y experimentado la robustez del algoritmo y sus múltiples oportunidades.

En un futuro se podría combinar la odometría VINS-Mono con las ventajas de otros sensores como el Lidar, proporcionando al vehículo aéreo las ventajas de las dos formas de odometría así como el mapeado del entorno con el Lidar.

8 CÓDIGOS

8.1 Código ejemplo de mundo en gazebo

```
<sdf version='1.6'>
<world name='default'>
 <light name='sun' type='directional'>
  <cast_shadows>1</cast_shadows>
  <pose frame=''>0 0 10 0 -0 0</pose>
  <diffuse>0.8 0.8 0.8 1</diffuse>
  <specular>0.2 0.2 0.2 1</specular>
  <attenuation>
   <range>1000</range>
   <constant>0.9</constant>
   <linear>0.01</linear>
   <quadratic>0.001</quadratic>
  </attenuation>
  <direction>-0.5 0.1 -0.9</direction>
 </light>
 <model name='ground_plane'>
  <static>1</static>
  <link name='link'>
   <collision name='collision'>
    <geometry>
     <plane>
      <normal>0 0 1</normal>
      <size>100 100</size>
     </plane>
    </geometry>
    <surface>
     <contact>
```

8.2 Código completo de MATLAB para creación de gráficas

```
%%Archivo para generación de gráficas resultados del TFG Guiado
%%visuo-inercial de vehículos autónomos
%%Autor: Germán Ferrando del Rincón
bag1=rosbag('gt2.bag')
bag2=rosbag('vins2.bag')
start1=bag1.StartTime;
end1=bag1.EndTime;

%%GT siempre empieza antes que VINS, por ello selecciono la parte en la que
%%ambos trabajan conjuntamente, eliminando datos carentes de valor
bag2select=select(bag2,'Time',[start1 end1])

%%Extraigo del mensaje de navegación la información que me interesa
vins_x=timeseries(bag1,'Pose.Pose.Position.X');
t_gtx=timeseries(bag2select,'Pose.Pose.Position.X');

vins_y=timeseries(bag1,'Pose.Pose.Position.Y');
t_gty=timeseries(bag2select,'Pose.Pose.Position.Y');

vins_z=timeseries(bag1,'Pose.Pose.Position.Z');
t_gtz=timeseries(bag2select,'Pose.Pose.Position.Z');


gt_x=t_gtx.Data;
gt_y=t_gty.Data;
gt_z=t_gtz.Data;
xvins=vins_x.Data;
yvins=vins_y.Data;
zvins=vins_z.Data;
%%Calculo los tiempos para poder interpolar   posteriormente
tvins=vins_x.Time;
tgt=t_gtx.Time;
%%Represento la evolución de la comparación entre VINS-Mono y Ground truth
figure(1)
subplot(3,1,1);
plot(vins_x, 'LineWidth', 1);
hold on;
plot(t_gtx, 'LineWidth', 1);
```

8.2 Código completo de MATLAB para creación de gráficas

```
title("Posición en x");
ylabel("(m)");
xlabel("Tiempo(s)");
legend('VINS-Mono','GT');
subplot(3,1,2);
plot(vins_y, 'LineWidth', 1);
hold on;
plot(t_gty, 'LineWidth', 1);
title("Posición en y");
ylabel("(m)");
xlabel("Tiempo(s)");
legend('VINS-Mono','GT');
subplot(3,1,3);

plot(vins_z, 'LineWidth', 1);hold on;
plot(t_gtz, 'LineWidth', 1);
title("Posición en z");
ylabel("(m)");
xlabel("Tiempo(s)");
legend('VINS-Mono','GT');
figure(2)
plot3(gt_x, gt_y,gt_z,'c*');
hold on;
plot3(xvins,yvins,zvins,'r.');
grid;
legend('VINS_Mono','GT');
ylabel("Eje y (m)");
xlabel("Eje x(m)");
zlabel("Eje z(m)");
```

8.2 Código completo de MATLAB para creación de gráficas

```
s=length(tvins);

r=length(tgt);
%%Interpolo para obtener el mismo número de muestras y así poder calcular
%%el error mediante la resta en valor absoluto
for i=1:s
   for x=1:r
      if (abs(tvins(i)-tgt(x))<0.09)
         gt_x1(i,1)=gt_x(x,1);
         gt_y1(i,1)=gt_y(x,1);
         gt_z1(i,1)=gt_z(x,1);

      end

   end

end
gt_x1=gt_x1(1:s);
gt_y1=gt_y1(1:s);
gt_z1=gt_z1(1:s);


error_x=abs(xvins-gt_x1);error_y=abs(yvins-gt_y1);error_z=abs(zvins-gt_z1);

error_2=sqrt(error_x.^2+error_y.^2+error_z.^2);
figure(3);
plot(tvins,error_2);
title('Error cuadrático');
ylabel('(m)');
xlabel('Tiempo(s)');
```

8.3 Código control.py

```
#!/usr/bin/env python

import numpy as np
import random
import math
import rospy
import roscpp
from   std_msgs.msg       import Float64
from   geometry_msgs.msg  import Twist
from   geometry_msgs.msg  import PoseStamped
from   nav_msgs.msg       import Odometry

x_current = 0.0
y_current = 0.0
z_current = 0.0


pos_obj=[0, 0]




def subscriptor(data):
   global x_act, y_act, z_act

   x_act  = data.pose.pose.position.x
   y_act  = data.pose.pose.position.y
   z_act  = data.pose.pose.position.z



def main():

   global  x_current,y_current,z_current


   kp_z   = 0.2
   kp_x= 0.2
```

8.3 Código control.py

```
        rospy.init_node('control_hector', anonymous=True)
        rate = rospy.Rate(10) # 5hz


        sub = rospy.Subscriber('/vins_estimator/odometry', Odometry, subscriptor)
        pub_cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size = 15)
        cmd =Twist()



    while not rospy.is_shutdown():

        print "Introduzca la coordenada z"
        valor=raw_input()
        pos_obj[0]=float(valor)
        dz=pos_obj[0]-z_act
        while dz>0.8:
          cmd.linear.z=kp_z*dz
          dz=pos_obj[0]-z_act
          pub_cmd_vel.publish(cmd)
          rate.sleep()


        pos_obj[1]=7
        dx=pos_obj[1]-x_act
        while dx>0.8:
            cmd.linear.x=kp_x*dx
           dx=pos_obj[1]-x_act
           pub_cmd_vel.publish(cmd)
           rate.sleep()


    rate.sleep()
    rospy.loginfo("Node is shutting down")

    # rospy.spin()

if __name__ == '__main__':
    main()
```

8.4 Código control_PID.py

```
#!/usr/bin/env python

import numpy as np
import random
import math
import rospy
import roscpp
from   std_msgs.msg      import Float64
from   geometry_msgs.msg  import Twist
from   geometry_msgs.msg  import PoseStamped
from   nav_msgs.msg      import Odometry
from  hector_uav_msgs.msg import PoseActionGoal
from   tf.transformations import euler_from_quaternion
from sensor_msgs.msg import LaserScan

x_act = 0.0
y_act = 0.0
z_act = 0.0

q_x=0.0
q_y=0.0
q_z=0.0
q_w=0.0


roll=0.0
pitch=0.0
yaw=0.0
matriz = np.empty((3,3))
deteccion=0
pi=3.14

def detector(laser):
        global deteccion
        u=len(laser.ranges)

        for i in range(u):

                angulo_rayo=laser.angle_min+(i*laser.angle_increment)
                distancia=laser.ranges[i]
                if  ((distancia<1) and (distancia>0.1))and (abs(angulo_rayo)<pi/4):
                        deteccion=1
```

8.4 Código control_PID.py

```
def subscriptor(data):
   global x_act, y_act, z_act, q_x, q_y, q_z, q_w, roll, pitch, yaw

   x_act  = data.pose.pose.position.x
   y_act  = data.pose.pose.position.y
   z_act  = data.pose.pose.position.z

   q_x=data.pose.pose.orientation.x
   q_y=data.pose.pose.orientation.y
   q_z=data.pose.pose.orientation.z
   q_w=data.pose.pose.orientation.w

   quaternions=[q_x, q_y, q_z, q_w]

   (roll, pitch, yaw) = euler_from_quaternion (quaternions)

def main():

   global  x_act,y_act,z_act,deteccion




   rospy.init_node('control_hector', anonymous=True)
   rate = rospy.Rate(10) # 5hz

   subdos=rospy.Subscriber('/scan',LaserScan,detector)

   sub = rospy.Subscriber('/ground_truth/state', Odometry, subscriptor)

   pub_cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size = 15)
   pub=rospy.Publisher('action/pose/goal', PoseActionGoal, queue_size=10)
   pos=PoseActionGoal()
   cmd =Twist()
   kp_lin = 0.2
   kd_lin = 0.01
   ki_lin = 0.004
   aux=0
   kp_ang = 3.4
   kd_ang = 0.2
   ki_ang = 0.02
```

8.4 Código control_PID.py

```
while not rospy.is_shutdown():
    subdos=rospy.Subscriber('/scan',LaserScan,detector)
    print "Introduce 3 posiciones de referencia"
    datos=('Eje x', 'Eje y', 'Eje z')
    for i in range(3):
            for x in range(3):
                print "Introduzca las 3",datos[i]
                valor=raw_input()
                matriz[i,x]=float(valor)
     for s in range(3):
         rate.sleep()
         longitud_i = abs(math.sqrt(((matriz[1,s]-y_act) ** 2) + ((matriz[0,s] - x_act) **
    2)+((matriz[2,s] - z_act) ** 2)))
          cmd.linear.x  = 0.0
          cmd.linear.y  = 0.0
          cmd.linear.z  = 0.0
          longitud_ant =  longitud_i
          longitud_x = abs(math.sqrt((matriz[0,s]-x_act) ** 2))
          longitud_y = abs(math.sqrt((matriz[1,s]-y_act) ** 2))
          longitud_z = abs(math.sqrt((matriz[2,s]-z_act) ** 2))
          longitud_ant_x     =  longitud_x
          longitud_ant_y     =  longitud_y
          longitud_ant_z     =  longitud_z
          total_error_x =0.0
          total_error_y =0.0
          total_error_z =0.0
          ang_obj=0.0
          error_ang_ant = 0.0
          error_ang = 0.0
          error_ang_act=0.0
```

8.4 Código control_PID.py

```
            cmd.angular.z=0.0
            deteccion=0
            while (abs(longitud_i)> 0.5 and aux==0):
                  ang_obj = math.atan2(matriz[1,s]-y_act, matriz[0,s]-x_act)
                  error_ang_act   = ang_obj-yaw
                  angular_speed = (error_ang_act*kp_ang) + ((error_ang_act-
error_ang_ant)*kd_ang) + (error_ang*ki_ang)
                   error_ang_ant = error_ang_act
                   error_ang = error_ang + error_ang_act
                   longitud_i = abs(math.sqrt(((matriz[1,s]-y_act) ** 2) + ((matriz[0,s] - x_act) **
2)+((matriz[2,s] - z_act) ** 2)))

                  longitud_x= abs(math.sqrt((matriz[0,s]-x_act) ** 2))

                  longitud_y= abs(math.sqrt((matriz[1,s]-y_act) ** 2))

                  longitud_z= abs(math.sqrt((matriz[2,s]-z_act) ** 2))

                linear_speed_x  = (longitud_x * kp_lin) + ((longitud_x- longitud_ant_x)* kd_lin)
          + (total_error_x*ki_lin)

                linear_speed_y  = (longitud_y * kp_lin) + ((longitud_y- longitud_ant_y)* kd_lin)
          + (total_error_y*ki_lin)

              linear_speed_z  = (longitud_z * kp_lin) + ((longitud_z- longitud_ant_z)* kd_lin) +
          (total_error_z*ki_lin)
              if (matriz[0,s]-x_act)<0:
                    linear_speed_x=-linear_speed_x
              if (matriz[1,s]-y_act)<0:
                    linear_speed_y=-linear_speed_y
              if (matriz[2,s]-z_act)<0:
                    linear_speed_z=-linear_speed_z
```

8.4 Código control_PID.py

```
cmd.linear.x  = linear_speed_x
cmd.linear.y  = linear_speed_y
cmd.linear.z  = linear_speed_z
cmd.angular.z=  angular_speed
longitud_ant_x     =  longitud_x
longitud_ant_y     =  longitud_y
longitud_ant_z     =  longitud_z
total_error_x      = total_error_x +  longitud_x
total_error_y      = total_error_y +  longitud_y
total_error_z      = total_error_z +  longitud_z
pub_cmd_vel.publish(cmd)
rate.sleep()


if deteccion==1
 rospy.loginfo("Obstaculo detectado, vuelta al anterior waypoint. Introduzca un waypoint alcanzable")
  if s==0:
        pos.goal.target_pose.pose.position.x=0
        pos.goal.target_pose.pose.position.y=0
        pos.goal.target_pose.pose.position.z=0
       pos.goal.target_pose.header.frame_id='world'
       pub.publish(pos)
  else:
       pos.goal.target_pose.pose.position.x=matriz[0,s-1]
       pos.goal.target_pose.pose.position.y=matriz[1,s-1]
      pos.goal.target_pose.pose.position.z=matriz[2,s-1]
      pos.goal.target_pose.header.frame_id='world'
      pub.publish(pos)
   s=3
   aux=1
```

8.4 Código control_PID.py

```
            rate.sleep()
            rospy.loginfo("Trayectoria finalizada")



if __name__ == '__main__':
                main()
```

REFERENCIAS

[1] Qin, Tong and Shen, Shaojie, « Online Temporal Calibration for Monocular Visual-Inertial Systems » *2018*.

[2] Qin, Tong and Li, Peiliang and Shen, Shaojie , « VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator »*2018*

[3] Paquete hector_quadrotor https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor

[4] Paquete robot_localization https://github.com/cra-ros-pkg/robot_localization

[5] Tutorial URDF http://wiki.ros.org/urdf

[6] Zheng Huai, Guoquan Huang, « Robocentric Visual Odometry »2018

[7] Tutorial suscriptor http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29

[8] Tutorial dependencias http://wiki.ros.org/ROS/Tutorials/rosdep

[9] Shaozu Cao, Xiuyuan Lu, Shaojie Shen, « GVINS: Tightly Coupled GNSS-Visual-Inertial for smooth and consistent state estimation» 2021

[10] Tutorial rosbag http://wiki.ros.org/ROS/Tutorials/Recording%20and%20playing%20back%20data

[11] Tutorial gazebo http://gazebosim.org/tutorials?tut=build_world&cat=build_world

[12] Raul Mur-Artal, Juan D. Tardos, « Visual-inertial monocular SLAM with map reuse» *2016*

ÍNDICE DE CÓDIGOS