Universidad Autónoma de Ciudad Juárez

Instituto de Ingeniería y Tecnología
Departamento de Ingeniería Industrial y Manufactura
Maestría en Tecnología

“Vehículo Autónomo a Escala con Navegación Basada en Visión para Inspección Inteligente de Pavimentos”

Tesis para obtener el grado de Maestro en Tecnología

Ing. Héctor Eduardo Tovanche Picón

“Becado por el Consejo Nacional de Ciencia y Tecnología”

Asesor
Dr. Ángel Flores Abad

Ciudad Juárez, Chihuahua
Julio 2020

Resumen:

Este trabajo presenta el desarrollo de una plataforma móvil compuesta por un vehículo a escala y dos sistemas embebidos con el propósito de conocer la ubicación en el entorno de dicha plataforma utilizando odometría visual y otro con una red neuronal convolucional basada en la arquitectura Resnet, entrenada para clasificación de imágenes, lo que permite la detección de fallas en el pavimento. Dichos sistemas fueron integrados en el vehículo a escala para realizar dichas tareas de manera autónoma. Los sistemas implementados corren de manera embebida en la plataforma utilizando una computadora a bordo. Los entornos de programación utilizados en este desarrollo fueron Python, Pytorch y la integración en ROS. La activación de los sistemas propuestos es realizada mediante la tele operación utilizando una computadora externa la cual se comunica mediante Wi-Fi a la plataforma.

Palabras clave:

Robot móvil; Inteligencia artificial; Redes neuronales; SLAM; Navegación autónoma

Índice general

Índice de figuras

Índice de tablas

1. Introducción

1.1 Antecedentes

En los últimos años la presencia de los robots en aplicaciones de la vida cotidiana y lugares de trabajo ha tenido un incremento, en diversas áreas como la manufactura donde se utilizan robots colaborativos en conjunto con la realidad virtual para asistir en tareas (Matsas, Vosniakos, y Batras, 2018), en la agricultura, a través de la recolección autónoma de frutos (Capua, Nacional, y Moreyra, 2018), en la minería, para la localización de seres humanos después de desastres (Ming-hui, Sai, y Xing-ru, 2017), en medicina, donde son utilizados para asistir cirugías (Zanagnolo, Garbi, Achilarre, y Minig, 2017), en la educación, donde sirven de apoyo en escuelas desde nivel básico hasta posgrados (Emily y cols., 2016) entre otros campos. La robótica móvil, área de la robótica que ha tenido un mayor desarrollo, campo que tiene raíces en disciplinas de las ciencias como mecánica, eléctrica, ingeniería electrónica y computación (Siegwart y Nourbakhsh, 2004). Una de las características que ha permitido una mayor utilización de robots móviles, es su capacidad de navegar alrededor del entorno sin estar anclado a un lugar fijo (Hayet, Hatem, y Jilani, 2016).

La capacidad de los robots móviles de navegar autónomamente en ambientes desconocidos ya sea dinámicos o estáticos, ha permitido que sean utilizados en diversas tareas, desde su uso en plantas industriales (Priyandoko, Ming, y Achmad, 2017), en la que son utilizados para labores de inspección en tareas en espacios confinados o de alto riesgo, o como robots de reconocimiento de estructuras (Motsch, Benammar, y Bergeon, 2017), donde permiten realizar tareas repetitivas, que normalmente requerirían un esfuerzo significativo por parte del operador para reconstruir edificios en modelos generados a partir de diseño asistido por computadora (CAD, por sus siglas en inglés). Además de las áreas anteriormente mencionadas, existe demanda de esta rama tecnológica en campos comerciales (Mathiassen, 2016), médicos (Burgner-Kahrs, Rucker, y Choset, 2015), militares (Kimon P y Vachtsevanos, 2015) y civiles (Givigi, Freitas, y Beaulieu, 2018), abriendo oportunidades para la investigación en este campo.

Una de las tareas básicas en la navegación de robots móviles en interiores, es la localización y mapeo simultáneo (SLAM por sus siglas en inglés) (Brahmanage y Leung, 2017), el cual consiste en navegación y posicionamiento en el entorno, dicho problema ha sido eje de investigaciones a lo largo del tiempo (Zaffar, Ehsan, Stolkin, y Maier, 2018).
El Sistema de Posicionamiento Global (GPS por sus siglas en inglés) puede proveer coordenadas terrestres a un robot, sin embargo la implementación en interiores representa dificultades debido a las débiles señales de los satélites y el error en la señal (Wu y Ding, 2018). Otras herramientas comunes para la navegación en interiores, son los algoritmos SLAM (Ibramigov y Afanasyev, 2017; Zaffar y cols., 2018), sin embargo, dichos algoritmos han sido condicionados por las capacidades de hardware. En los años 2000 se utilizaban sensores láser (Brahmanage y Leung, 2017), cuya limitante es la representación en solo dos dimensiones, además del alto costo computacional y económico. Los sensores

de rango LIDAR (acrónimo en inglés de detección de luz y rango) han sido implementados en conjunto a algoritmos SLAM, sin embargo, su costo superior a 6000 dólares (Bergerman y cols., 2015), lo convierte en una solución costosa.

Dentro de los sentidos que pueden ser adquiridos por los robots móviles mediante la implementación de sensores, se puede encontrar la visión, la cual se refiere a la capacidad de un robot de percibir visualmente el entorno y utilizar esta información al ejecutar diferentes tareas (Kragic y Vincze, 2009). La retroalimentación visual ha sido utilizada para la navegación autónoma y evasión de obstáculos (Endres y cols., 2012).

Además del problema de localización y mapeo, otro problema relacionado en la navegación autónoma en un robot móvil es guiar el movimiento según trayectorias (Aamer y Ramachandran, 2015) generadas a partir de la información proveniente de sensores externos (Bambino, 2008), como lo es un sensor de visión RGB-D. En los últimos años con la irrupción de cámaras de profundidad como son los sensores RGB-D (Mapa de color Rojo, Verde, Azul por sus siglas en inglés) (Endres y cols., 2012) capaces tanto de medir como proporcionar información visual, aunado a su bajo costo (Motsch y cols., 2017), se ha potencializado la creación de algoritmos Visual-SLAM, basados en las técnicas clásicas SLAM con el propósito de resolver el problema de localización y mapeo simultáneo con la integración de un sensor de visión (Capua y cols., 2018).
Los sensores tipo RGB-D son el resultado de la combinación de cámaras monoculares, transmisores y receptores infrarrojos (Zaffar y cols., 2018). Ocasionando que la cantidad de información generada, la configuración de cuadros por segundo a baja latencia del sensor y la implementación conjunta de algún algoritmo SLAM, cree un gasto computacional elevado (Kerl, Sturm, y Daniel, 2013).

Otra de las ramas basada en la percepción visual en robots que ha tenido gran auge es la visión por computador, gracias a los avances de las técnicas de aprendizaje profundo (Kumra y Kanan, 2017), dentro de las aplicaciones en las que convergen la robótica y el aprendizaje profundo, encontramos avances en la medicina a través de la implementación de asistentes robóticos para cirugía (Shvets, Rakhlin, Kalinin, y Iglovikov, 2019), mejoras en los brazos robóticos para la sujeción de objetos (Levine, Pastor, Krizhevsky, Ibarz, y Quillen, 2018) y la solución de pruebas de Turing mejor conocidas como captchas utilizando estos conceptos (Sivakorn, Polakis, y Keromytis, 2016). Las técnicas de aprendizaje profundo son definidas por (Lecun, Bengio, y Hinton, 2015), como métodos de representación del aprendizaje con múltiples niveles de representación a través de la integración simple y no lineal de módulos en la cual cada módulo transforma la representación en un nivel (Lecun y cols., 2015). El aprendizaje profundo se clasifica como un sub-campo de aprendizaje de máquinas, que a su vez es un sub-campo de la inteligencia artificial (Rosebrock, 2017). En paralelo, uno de los principales problemas de movilidad de ciudad Juárez, radica en las fallas en asfalto (Gamboa, 2019; Juarez, 2019; Olmos, 2020), problema que ha estado presente en la localidad a lo largo de la última década, las problemáticas presentes en la operación y utilización de asfalto es el deterioro en función del tiempo, la flujo de tráfico y variaciones ambientales (Gopalakrishnan, Khaitan, Choudhary, y Agrawal, 2017). La detección de grietas en el pavimento se ha convertido en una tarea critica en el monitoreo e inspección

de estructuras de ingeniería civil (Dung y Anh, 2019). Técnicas de monitoreo manuales son dificultadas debido a limitaciones espaciales y riesgos presentes en el entorno además del tiempo-hombre necesario para ser llevados a cabo (Shehata, Mohamed, Abdellatif, y Awad, 2018).

1.2 Planteamiento del problema

El deterioro del pavimento en las ciudades ocurre de forma natural al pasar del tiempo, situaciones ambientales como elevadas temperaturas o lluvia aceleran este deterioro, por lo tanto suelen ocasionar problemas diversos, de movilidad, daño en los vehículos, provocar accidentes viales, entre otros. Actualmente la inspección asfáltica se lleva a cabo de forma manual, con largos tiempos de recorrido, incluye una dosis de peligro para el inspector que a la vez esta sujeto a errores de inspección por parte del personal. Por lo cual es deseable contar con tecnología autónoma para realizar la tarea de inspección y proporcionar un método más sistematizado.

1.3 Objetivos

1.3.1. Objetivo general

Desarrollar un sistema de inspección inteligente para la detección de fallas en el pavimento y un sistema de navegación autónoma.

1.3.2. Objetivos específicos

- Seleccionar el robot móvil para el desarrollo de la aplicación en función de las necesidades de carga del sistema.
- Configurar e integrar sensores de visión.
- Desarrollar un algoritmo de navegación basado en la odometría enviada por el sensor T265.
- Entrenar una red neuronal para detección de fisuras y agrietamientos en el asfalto.
- Integrar red neuronal entrenada un robot móvil para detección autónoma de fallas en el pavimento.
- Validar operación, reconocimiento y navegación.

1.4 Hipótesis

1.4.1. Hipótesis general

El desarrollo de un sistema autónomo inteligente de inspección de asfalto, permitirá detectar fallas en un área o sección deseado. Dicha inspección puede ser realizada de manera autónoma dotando de capacidades sensoriales a un robot móvil, integrando un algoritmo de navegación basado en V-SLAM, para permitir la navegación autónoma en espacios cerrados y desconocidos.

1.5 Justificación

Conocer con precisión fallas tempranas en el pavimento tales como grietas, sin requerir recursos humanos, permite agilizar y disminuir los costes de estas operaciones. Los avances en ciencias computacionales tales como sistemas de visión e inteligencia artificial ha permitido que se puedan crear sistemas embebidos que realicen este tipo de actividades in situ. Por lo tanto, se propone la integración de dos algoritmos; uno de navegación basada en V-SLAM, capaz de ser implementado en una plataforma robótica para brindar de percepción del entorno y lograr tareas de navegación autónoma y otro basado en técnicas de aprendizaje profundo capaz de identificar fallas en asfalto. Demostrar que el implementar el binomio de algoritmos en un robot móvil comercial, puede ser empleado para resolver el problema de navegación autónoma y monitoreo de superficies, contribuyendo al desarrollo de actividades de mayor complejidad basadas en principios de navegación autónoma, sin la necesidad de adquirir hardware especializado con un costo elevado.

La implementación de este proyecto permitirá la generación de conocimiento para el uso futuro de robots móviles con integración de algoritmos de navegación autónoma tipo V-SLAM, en actividades que requieren un alto nivel de certidumbre respecto al entorno, cuando son llevadas a cabo en espacios cerrados. Así como en proyectos que requieran el reconocimiento de características especificas del entorno a través de la aplicación de algoritmos de inteligencia artificial.

1.6 Alcances y delimitación

1.6.1. Alcances

- Desarrollo de un sistema de navegación basado en V-SLAM y un sistema inteligente de detección de fallas en el asfalto en un vehículo a escala.
- Integración de algoritmo de navegación V-SLAM en un vehículo a escala para navegación autónoma en entornos cerrados y desconocidos.
- Entrenamiento y generación de una red neuronal para la identificación de fallas en asfalto.
- Clasificación en tiempo real de fallas en el asfalto.
- Integración simultánea de ambos conceptos en un vehículo a escala.

1.6.2. Delimitaciones

- Utilización de una cámara para la adquisición de la localización respecto del entorno.
- El tiempo de navegación autónoma estará limitado por la capacidad de la batería del vehículo.
- La identificación de fallas en el asfalto estará limitado a grietas.
- Las condiciones de iluminación afectaran el desempeño de ambos sistemas.
- El vehículo será a escala 1/10.

2. Marco teórico

En este capítulo se definirán los elementos principales que integran este proyecto de investigación. Además, se definen conceptos de gran importancia sobre los cuales se desarrolla este proyecto.

2.1 Procesamiento de una imagen

2.1.1. Píxel

La mínima representación de una imagen es llamada píxel, el píxel compuesto por información que indica el color o la intensidad de luz presente en un lugar de la imagen (Rosebrock, 2017), si se toma una imagen como una matriz de n x n, cada elemento de dicha matriz será un píxel. En la figura 2.1 se presenta una imagen compuesta por 1000 píxeles horizontales y 750 píxeles verticales, es decir, 1000 x 750 píxeles = 750,000 píxeles totales.

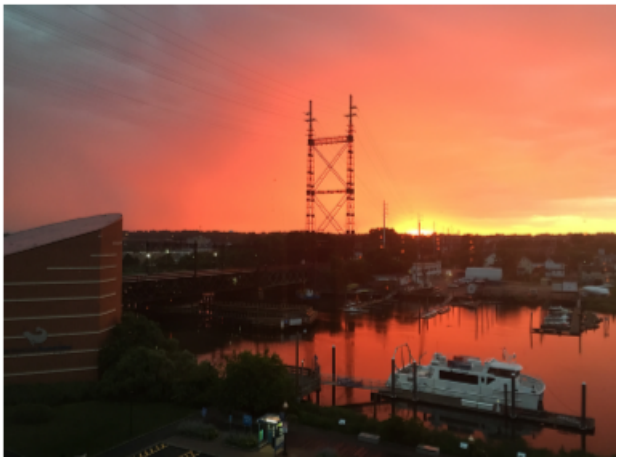


Figura 2.1: Imagen de 1000 x 750 píxeles (Rosebrock, 2017)

2.1.2. Espacio de color

Un píxel puede contener información del color de esa sección de la imagen en tres maneras:

- Binario, representación en blanco y negro con un solo canal, figura 2.2.
- Escala de grises, un solo canal con rango 8 bits, es decir de 0 a 255, donde 0 corresponde a negro y 255 a blanco, figura 2.3.
- Representación a color, la cantidad de canales varía en función del espacio de color utilizado, comúnmente se utiliza el espacio de color RGB, compuesto por 3 canales, rojo, verde y azul, figura 2.4.

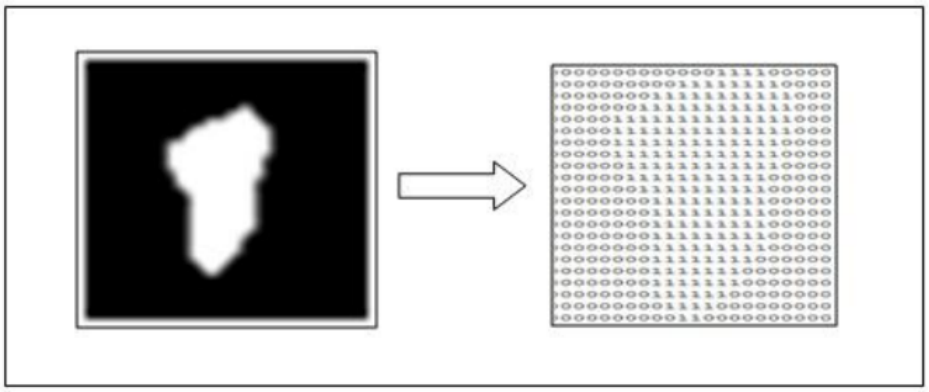


Figura 2.2: Representación binaria de una imagen (Ab Ghani y cols., 2011)

Figura 2.3: Representación a escala de grises de dos imágenes (Shnayderman y cols., 2006)

Figura 2.4: Representación de una imagen en 3 canales de color (RGB) (Rosebrock, 2017)

2.1.3. Sistema coordenado de una imagen.

Como se mencionó anteriormente, una imagen está compuesta por un arreglo matricial de píxeles, con origen en el píxel 0,0. Con forma x, y, la figura 2.5 muestra el inicio de la letra I en la coordenada 0,0 y el final de la letra I en la coordenada 7,7.

2.2 Convoluciones

2.2.1. Filtros

En procesamiento de imágenes un filtro puede ser creado utilizando una matriz de convolución (Shapiro y Stockman, 2001), la cual utiliza la matriz de la imagen original y la multiplica por otra matriz, llamada kernel, la cual modifica el efecto que el filtro tendrá en la imagen. Comúnmente son utilizados matrices o kernels de 3x3 para el procesamiento de imágenes, en la figura 2.6, se presenta un kernel de 3x3 utilizado para crear un efecto de blur en la imagen. La figura 2.7 presenta el resultado de aplicar un kernel que amplifica los bordes verticales en una imagen. Estos ejemplos son de importancia

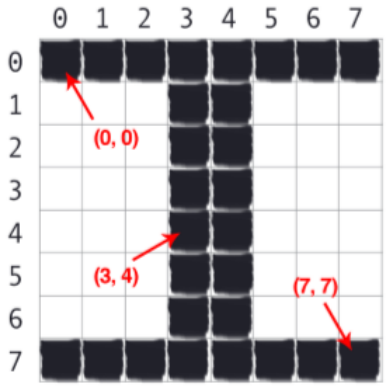


Figura 2.5: Letra I en un plano matricial x, y (Rosebrock, 2017)

en el pre-tratamiento de una imagen, ya que permiten encontrar características de una imagen, a pesar de eliminar información de la misma, permitiendo un procesamiento de la imagen más ágil.

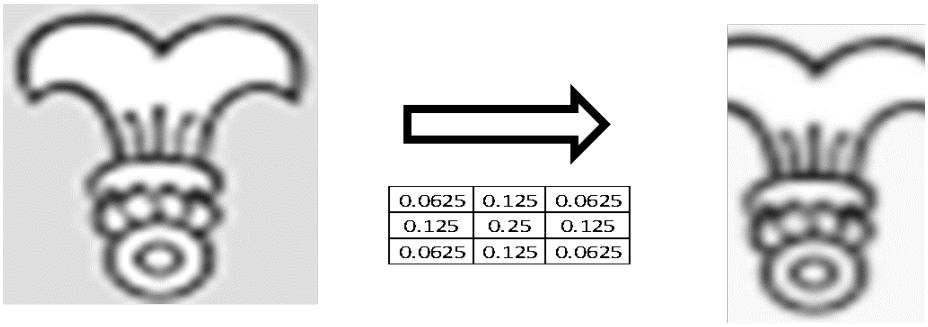


Figura 2.6: Filtro blur (Powell, 2018)

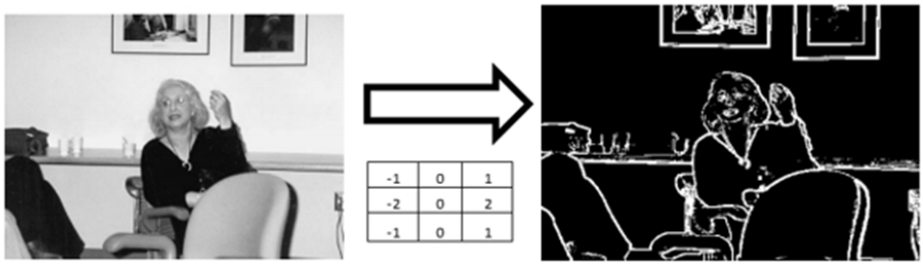


Figura 2.7: Filtro bordes verticales (Shapiro y Stockman, 2001)

2.3 Inteligencia artificial

La inteligencia artificial o también conocida como programación heurística es una rama de la ciencia de la informática enfocada en resolver problemas que para el ser humano son intuitivos y fáciles de resolver, pero para un ordenador suponen un reto mayor. El objetivo de esta rama es dotar a los ordenadores de habilidades para la resolución de problemas tales como la toma de decisiones,

percepción, entendimiento de la comunicación humana, resolución de problemas desde la perspectiva intelectual del ser humano (Findler, 2007).

2.3.1. Aprendizaje automático (Machine learning)

El aprendizaje automático que data desde los años 60, es una sub-disciplina de la inteligencia artificial, la cual busca utilizar algoritmos basados en modelos numéricos para la resolución de problemas, de acuerdo a Arthur Samuel, el aprendizaje automático, le brinda a los ordenadores la capacidad de aprender sin haber sido programadas con anterioridad (Samuel, 1959). La clasificación de los tipos de algoritmos para aprendizaje automático es la siguiente:

- Aprendizaje supervisado, la computadora es entrenada con información identificada, algoritmos comunes:
  - Redes neuronales.
  - Regresión lineal, figura 2.8.
  - Clasificador Naive Bayes.
  - Arboles de decisión.
- Aprendizaje no supervisado, la computadora es entrenada con información no identificada, la computadora buscara encontrar patrones en la información disponible, algoritmos comunes:
  - Reglas de asociación.
  - Clusterización por k-media.
- Aprendizaje por reforzamiento, el entorno es evaluado por el algoritmo, una función de tomas de decisión es usada para realizar una acción, al algoritmo evalúa el efecto de dicha acción en el entorno, ejemplos de algoritmos:
  - Q-Aprendizaje.
  - Diferencia temporal.
  - Redes profundas adversarias.

2.3.2. Red neuronal artificial

Una red neuronal artificial o sistema conexionista es un tipo de arquitectura de aprendizaje automático basada en el modelo biológico de una neurona humana. Dicho modelo está construido por un conjunto de unidades llamadas neuronas, las cuales están interconectadas entre sí, trasmitiendo información entre ellas. El valor de entrada de cada neurona es multiplicado por un valor dinámico llamado Peso. Una red neuronal está compuesta por 3 capas o niveles, entrada, capa intermedia y salida. La representación gráfica de este proceso se visualiza en la figura 2.9, donde las tres capas de una red neuronal están representadas a través de círculos interconectados.

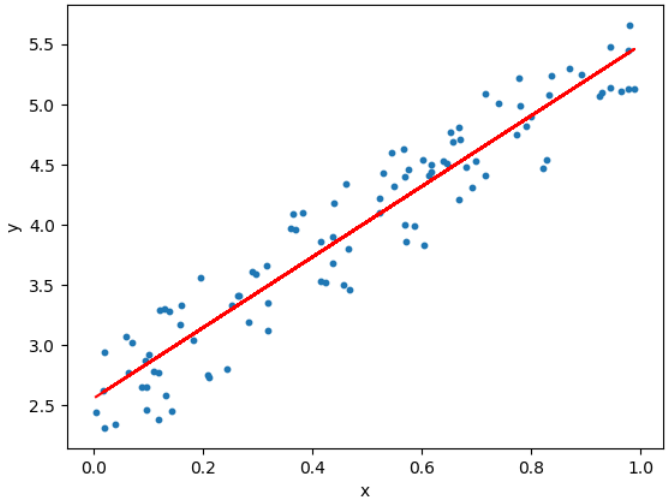


Figura 2.8: Gráfica de regresión lineal en Python (Rubiales, 2019)

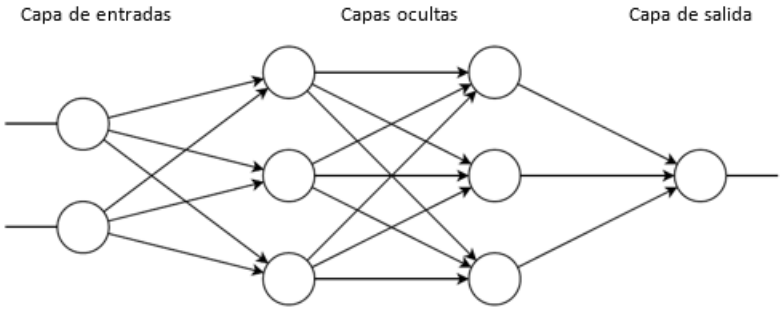


Figura 2.9: Red neuronal genérica

2.3.3. Aprendizaje profundo

Aprendizaje profundo o Deep learning, es una técnica de aprendizaje automático que utiliza más de una red neuronal artificial, integrada, además, de niveles jerárquicos. Esta configuración permite aprender representaciones de información con diferentes niveles de abstracción (Lecun y cols., 2015), aplicaciones como reconocimiento del habla, reconocimiento visual de objetos, detección visual, genómicos entre otros, han sido beneficiados con aplicaciones de este tipo. Un ejemplo de red genérica de este tipo es mostrado en la figura 2.10. La figura 2.11 presenta un mapa mental de esta rama de las ciencias computacionales.

2.4 Cámara de profundidad RGB-D

La mayoría de los sistemas de mapeo en 3D están integrados por 3 componentes principales. La proyección en el espacio de matrices de puntos consecutivos, la detección de distancia entre punto y punto y por último la alineación consistente global de la secuencia de puntos. La captura de nubes de puntos 3D y la reconstrucción de datos 3D está bien ejecutada las cámaras de profundidad con-

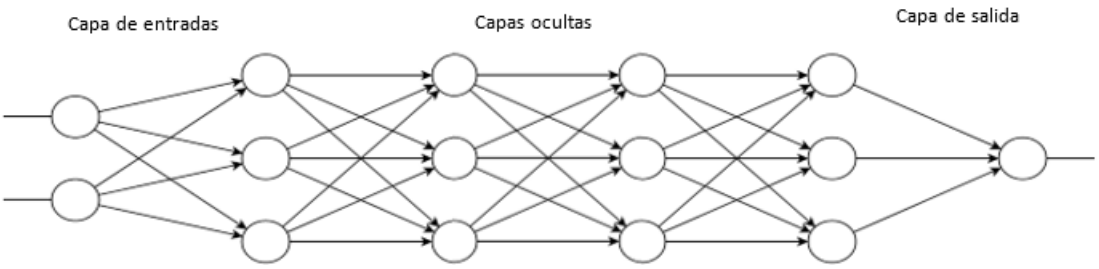


Figura 2.10: Red neuronal en aprendizaje profundo

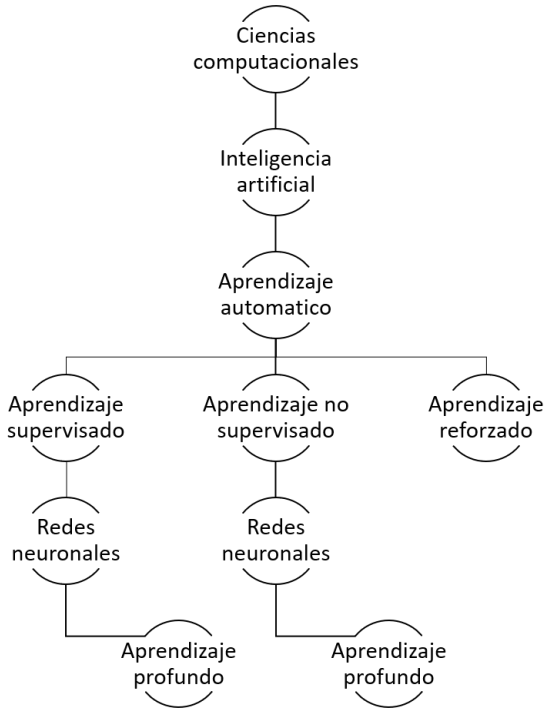


Figura 2.11: Mapa mental de la división de las ciencias computacionales

vencionales, sin embargo, ignoran valiosa información contenida en las imágenes RGB (Ren, Bo, y Fox, 2012). Las cámaras de color por otra parte capturan bastante información visual sin embargo esta extremadamente difícil extraer información de la profundidad de los objetos frente a la cámara solo con los datos que puede recabar especialmente en los ambientes cerrados con poca o nula iluminación. Las cámaras RGB-D son sistemas de detección que capturan imágenes RGB (Visual) junto con la información de profundidad de cada píxel. A pesar de que sistemas de detección con estas capacidades han sido fabricadas durante años es ahora cuando están siendo elaborados con características que los hacen atractivos para la investigación fuera del campo de visión computarizada. Las aplicaciones más comunes de estos sistemas son la investigación y entretenimiento en el hogar (Henry, Krainin, Herbst, Ren, y Fox, 2012). En la figura 2.12 podemos apreciar un ejemplo de cámara RGB-D, el modelo d435 de la gama Realsense de Intel. La figura 2.13 representa las salidas de este tipo de cámaras, la imagen convencional obtenida por el sensor RGB y la imagen de profundidad.

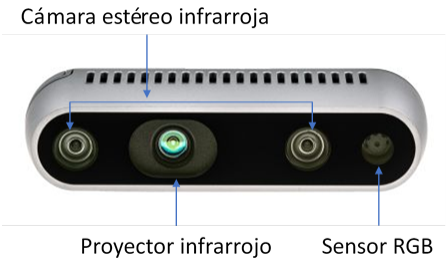


Figura 2.12: Componentes de cámara RGB-D (Intel, 2019a)

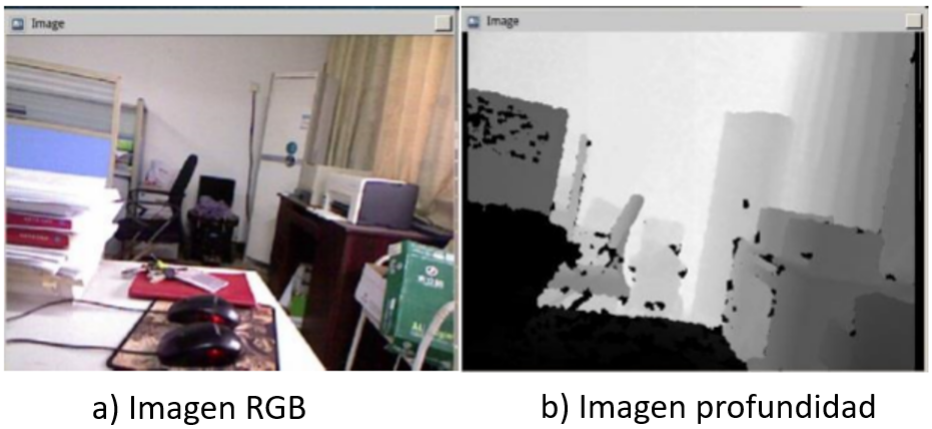


Figura 2.13: Salidas de cámara RGB-D (Liu y cols., 2017)

2.5 Robot móvil

Un robot móvil posee la capacidad de moverse en entornos variables y desconocidos de los que posee un conocimiento incierto, a través de la interpretación de la información obtenida a través de los sensores con los cual esta equipado y el estado actual del vehículo (Bambino, 2008).

2.5.1. Robots móviles terrestres

Los robots móviles terrestres conocidos como UGV (Unmanned ground vehicle), son equipos mecánicos capaces de moverse por superficies terrestres para cumplir con actividades de transporte o llevar objetos con una funcionalidad definida, donde específicamente no llevan un operador humano (Gage, 1995). Los vehículos terrestres no tripulados se pueden clasificar en función del tipo de control empleado, los robots operados remotamente, son aquellos donde un operador humano utiliza una interfaz remota para enviar comandos al robot. Aquellos vehículos terrestres no tripulados que operan sin contar con un operador humano son conocidos como UGV autónomos. El prototipo presentado en la figura 2.14, es un ejemplo de un robot móvil terrestre, desarrollado en la universidad de Innopolis con fines de investigación.

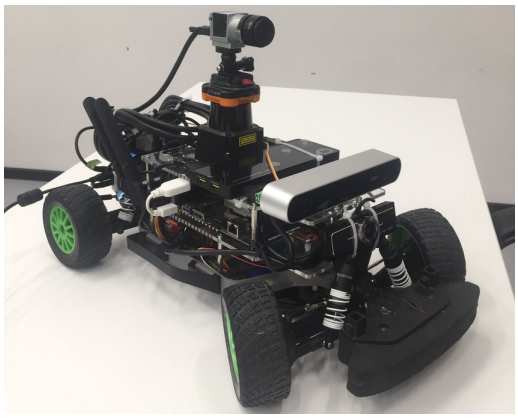


Figura 2.14: Prototipo UGV Universidad de Innopolis (Filipenko y Afanasyev, 2018)

2.6 Navegación autónoma

Los robots móviles operando en entornos variables y no conocidos se enfrentan a incertidumbres significativas en la posición e identificación de objetos en el entorno. Dicha incertidumbre es tal, que el traslado desde un punto "A" a un punto "B" es una actividad de riesgo para el robot móvil (Bambino, 2008; Víctor y cols., 2007). La navegación de robots móviles puede ser categorizada en tres subgéneros, navegación basada en mapas, donde le robot posee información a priori del mapa del entorno, navegación basada en construcción de mapas, donde el robot construye el mapa del entorno, por último, navegación sin mapas, el robot móvil no utiliza un modelo explícito y navega basado en la identificación de objetos (DeSouza y Kak, 2002). El problema SLAM, es un problema de la navegación (Endres y cols., 2012; Ortiz, Yu, y Zamora, 2019) basado en construcción de mapas.

2.7 Marcos de referencia

Un sistema de coordenadas o referencia define la posición de un robot respecto a otro elemento con un sistema de coordenadas dado, por lo que la posición de un robot en un sistema de coordenadas respecto a un marco de referencia dado puede ser representada como el vector $\mathbf{p}$ de características $\mathbf{R} \in \mathbb{R}^{3\times 1}$ (Vazquez Cespedes, 2011). El vector $\mathbf{p}$ en un marco de referencia $\{E\}$ es mostrado en la ecuación 2.1.

$$\mathbf{p}^E = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{2.1}$$

Existen dos convenciones para determinar el marco de referencia cuando se representan vectores de estado en la navegación. Dichas convenciones son:

1. NED (North, East, Down), comúnmente utilizado en aeroespacial, figura 2.15 donde el marco referencia $\{E\}$ representa el marco de referencia inercial y $\{B\}$ representa el marco de referencia local.

2. ENU (East, North, Up) utilizado en la geografía, figura 2.16 donde el marco referencia $\{E\}$ representa el marco de referencia inercial y $\{B\}$ representa el marco de referencia rígido.

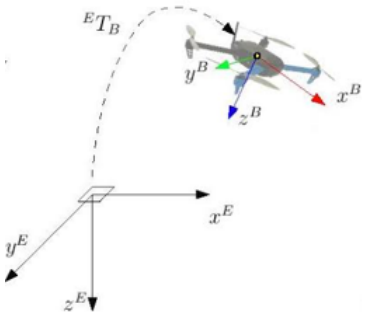


Figura 2.15: Marco de referencia NED (autopilot, 2019)

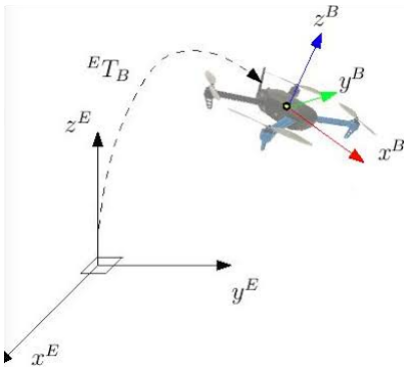


Figura 2.16: Marco de referencia ENU (autopilot, 2019)

2.7.1. Representación matemática de la orientación

Se puede representar la orientación de un objeto respecto a un marco de referencia fijo utilizando los ejes coordenados que representan el marco de referencia del objeto (Wang y Wilson, 1992), la orientación del objeto puede ser representada matemáticamente de diversas formas, una de ellas son los ángulos RPY comúnmente usados en navegación, dichos ángulos son representados de la siguiente manera (Onal, 2008):

1. Cabeceo (Pitch) rotación respecto al eje ala-ala, eje coordenado y.

2. Alabeo (Roll) rotación respecto al eje morro-cola, eje coordenado x.

3. Guiñada (Yaw) rotación respecto al eje vertical, eje coordenado z.

Estos ángulos son visibles en la figura 2.17.

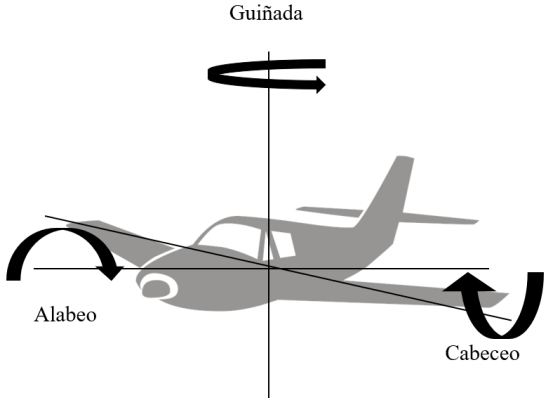


Figura 2.17: Rotaciones en navegación

2.8 Matriz de transformación homogénea

La matriz de transformación homogénea permite representar la rotación y traslación de un objeto respecto un marco de referencia fijo (Corke, 2017). Para determinar la nueva posición de un objeto es necesario utilizar una matriz que expresa la orientación y traslación del marco $\{\mathrm{B}\}$ respecto al marco de referencia inercial $\{\mathrm{A}\}$, misma que es denominada matriz de transformación homogénea 2.2, donde $\boldsymbol{t} \in \mathbb{R}^3$ es un vector que define el origen del marco $\{\mathrm{B}\}$ respecto al marco $\{\mathrm{A}\}$, y $\boldsymbol{R}$ es la matriz ortonormal de 3×3 quien describe la orientación de los ejes del objeto (marco $\{\mathrm{B}\}$), respecto al marco de referencia inercial $\{\mathrm{A}\}$. La figura 3.18 representa la relación de los marcos de referencia $\{\mathrm{B}\}$ y $\{\mathrm{A}\}$.

$$ {}^{A}\mathbf{T}_{B} = \begin{bmatrix} {}^{A}\mathbf{R}_{B} & \mathbf{t} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix} = \begin{bmatrix} {}^{A}\mathbf{R}_{B3\times 3} & \mathbf{t}_{3\times 1} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix} \tag{2.2} $$

2.9 SLAM

Localización y mapeo simultáneo también conocido como SLAM, es un problema fundamental de la robótica (Henry y cols., 2012; Ortiz y cols., 2019), el cual se basa en la problemática de la necesidad de construir un mapa del entorno mientras se determina la localización del robot dentro del mapa. Por lo tanto, SLAM es el proceso en el cual un robot móvil puede construir un mapa del entorno y al mismo tiempo usar este mapa para deducir su localización. En principio, tanto la ubicación del robot y el mapa son desconocidos, el vehículo posee un modelo cinemático y se mueve a través del entorno desconocido, el cual posee marcas artificiales o naturales. El problema SLAM se basa en encontrar la representación apropiada de la observación y los modelos de movimiento (Aulinas, Petillot, Salvi, y Lladó, 2008). El marco matemático de un algoritmo SLAM está basado en un proceso de estimación en el cual, con un modelo cinemático/dinámico dado del vehículo y la observación relativa entre el vehículo y las marcas en el entorno, se estima la estructura del mapa y la posición del vehículo, la velocidad y la orientación dentro del mapa (Kim y Sukkarieh, 2004). La figura 2.18 muestra la salida de un mapa obtenido a través de un sistema tipo SLAM. Los sistemas SLAM pueden ser categorizados en 4 categorías en función del tipo de sensor implementado (Chen, Samarabandu, y Rodrigo, 2007),

las categorías son:

- Sistemas basados en Laser.
- Sistemas basados en Sonar.
- Sistemas basados en Inercia.
- Sistemas basados en Visión.

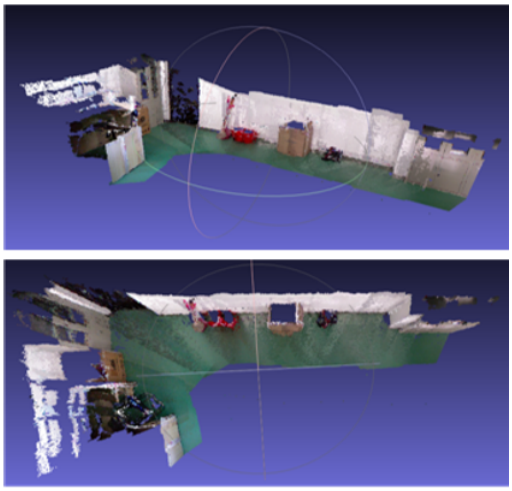


Figura 2.18: Mapa 3D construido con técnica SLAM (Li y cols., 2017)

2.9.1. Sistema basado en visión V-SLAM

Las técnicas de SLAM basadas en sensores de visión se han popularizado gracias a sus características tales como un rango de medición alto, alta resolución de los sensores y sus propiedades pasivas, es decir al no emitir energía posibilitan la integración conjunta de otros sensores (Chen y cols., 2007). El funcionamiento de un algoritmo SLAM basado en sensores de visión posee dos procesos simultáneos, el robot en el que este implementado partirá de un punto desconocido de un mapa desconocido, al moverse la odometría cambiará y se actualizará. Simultáneamente los datos adquiridos por el sensor de visión son procesados y se extraen puntos característicos en el entorno, a continuación, se determina si los puntos característicos han sido observados con anterioridad o son nuevos, en caso de existir un punto conocido se introduce al algoritmo como un re-observación, si se trata de un punto nuevo se define una nueva observación que será utilizada para actualizar la estimación de la posición (Capua y cols., 2018; Lopez Torres, 2016), este proceso se puede apreciar en la figura 2.19.

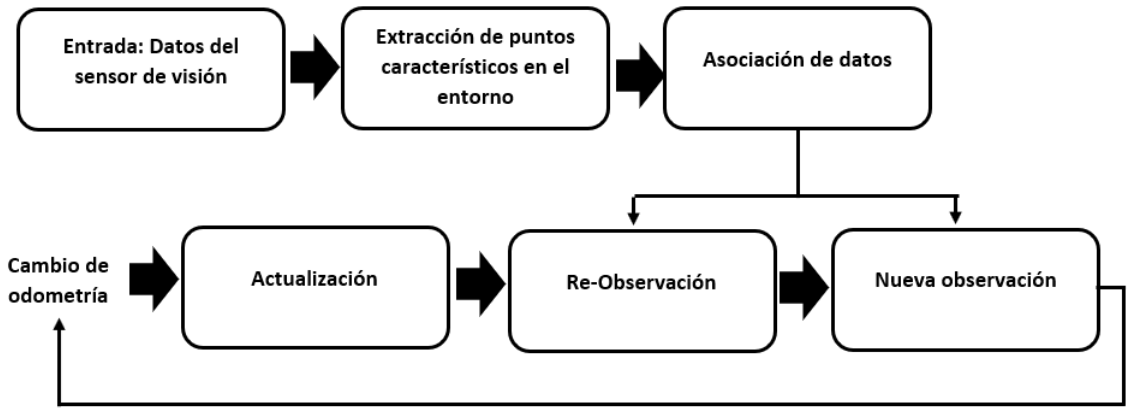


Figura 2.19: Esquema de algoritmo SLAM basado en visión

2.10 Software

En esta sección se presenta el software utilizado en la plataforma, así como sus particularidades, se incluye desde el sistema operativo seleccionado, ambiente de programación, software específico y librerías utilizadas.

El software utilizado está distribuido en dos vertientes, el software requerido para la implementación de un sistema de navegación autónomo basado en visión y el software utilizado para implementar un sistema de renacimiento basado en aprendizaje profundo. Todo a partir del mismo sistema operativo, una distribución de Linux llamada Ubuntu, corriendo en la plataforma Jetson Nano. Para las cuestiones de navegación del robot, se eligió el software de uso libre Ardupilot, el cual actúa como el sistema operativo del robot, una representación visual de esto puede observarse en la figura 2.20.

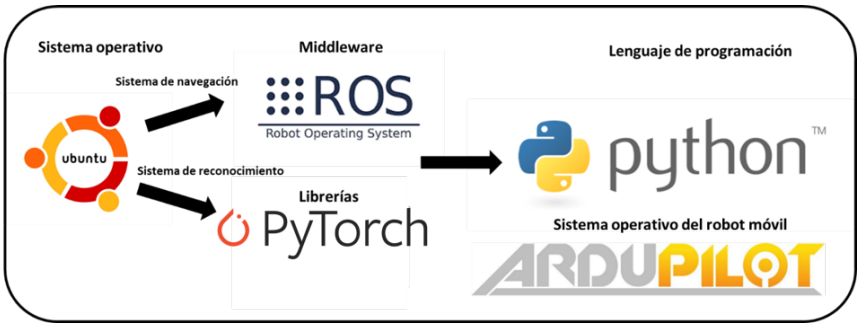


Figura 2.20: Mapa mental del entorno de programación

2.10.1. Ubuntu 18.04

Ubuntu es una distribución de Linux, software de código libre, es implementado como sistema operativo y está basado en Debian, la plataforma Jetson Nano, permite su utilización de manera directa e intuitiva, debido a las características de software libre es de acceso gratuito. La figura 2.21 nos muestra el escritorio de Ubuntu 18.04 corriendo en un Jetson Nano.

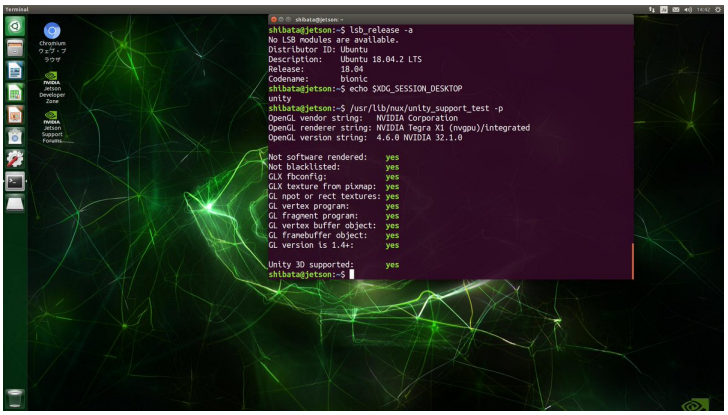


Figura 2.21: Escritorio Ubuntu en un Jetson Nano

2.10.2. Robotic Operating System (ROS)

ROS es definido por sus creadores como un framework o middleware de desarrollo de software para robots, creado en el 2007 con el nombre de Switchyard, en el laboratorio de inteligencia artificial de la universidad de Standford. Dentro de las bondades de este sistema se encuentra la posibilidad de comunicar software de alto nivel con hardware, la integración de múltiples lenguajes de programación como C++ y Python, además ROS posee múltiples herramientas para simulación, todo esto con la característica de ser Software libre. En cuanto al gasto computacional, ROS no consume muchos recursos, por lo cual su implementación en sistema embebidos con bajas capacidades computacionales es posible. ROS funciona bajo dos conceptos principales, suscripción/publicación y los servicios, el mecanismo de comunicación empleado está basado en el envío y recepción de mensajes, la información es publicada bajo tópicos (topics) asociados a un tipo de mensaje concreto. Los servicios permiten la conexión entre dos elementos de manera exclusiva. Un tercer elemento, llamado Nodo, permite constante comunicación con el exterior figura 2.22. Un elemento del exterior puede ser cualquier actuador conectado al sistema, un servomotor, un indicador visual entre otros.

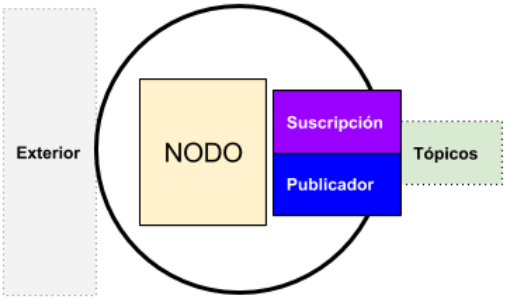


Figura 2.22: Comunicación en ROS

2.10.3. Python

Python es un lenguaje de programación desarrollado por Guido van Rossum en 1991, a través de los años ha recibido constantes actualizaciones, llegado a la última versión estable 3.8.2, en la implementación de este proyecto es utilizada la versión 3.5. La filosofía de este lenguaje de programación está basada en la legibilidad del código es un lenguaje de programación multiparadigma debido a su soporte a la orientación de objetos. Este lenguaje de programación posee una licencia de código abierto, por lo cual su uso es de carácter gratuito. En la actualidad ha incrementado su popularidad en la investigación y desarrollo, debido al soporte de grandes empresas, así como de la comunidad. Python puede ser utilizado en los sistemas operativos Mac, Windows y Linux, la figura 2.23 muestra el entorno de programación Python 3.6 en Ubuntu.

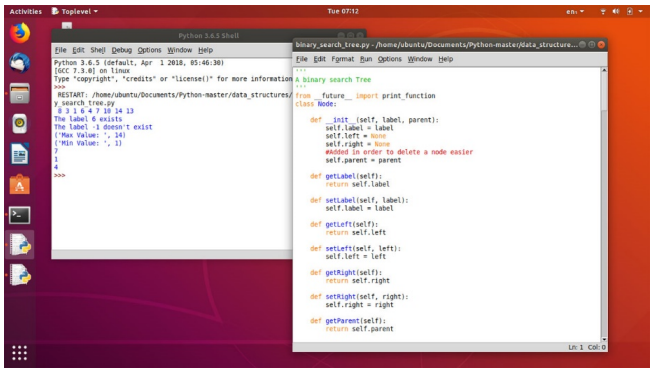


Figura 2.23: Python 3.6 IDE

2.10.4. PyTorch

PyTorch es una librería de machine learning de código abierto, basado en la librería Torch para su aplicación en lenguaje Python, es utilizado en áreas como visión por computadora y procesamiento natural del lenguaje, PyTorch posee dos características de alto nivel, el computo de tensores con soporte de procesamiento a través de unidades de procesamiento gráfico (GPU) y la construcción de redes neuronales profundas basadas en sistemas de diferenciación automática.

2.10.5. Ardupilot

Ardupilot es un software de código abierto para la implementación en vehículos no tripulados, capaz de controlar autónomamente drones multirotor, aviones de ala fija, helicópteros, UGV, botes, submarinos entre otros. Originalmente desarrollado por entusiastas se ha convertido en una plataforma confiable con gran cantidad de características utilizada en la industria y organizaciones investigadoras. La plataforma está integrada por dos elementos, el software de navegación, embebido en la plataforma móvil, comúnmente conocido como firmware. Además de una estación de control terrestre tal como, Mission Planner, APM planner, QGroundControl, MaxProxy, Tower, entre otros. La conjunción de ambos elementos permite la implementación de un sistema autónomo en un robot móvil. En la figura 2.24 se muestra la estación de control terrestre Mission Planner.

Figura 2.24: Mission planner

3. Desarrollo de sistema de navegación basado en visión

El sistema propuesto en este trabajo será el encargado de realizar las actividades de inspección autónomas, por lo cual deberá integrar actuadores y sensores que cumplan con los requerimientos necesarios para el desarrollo de los sistemas mostrados en la figura 3.1. La integración de la plataforma y el sistema de navegación serán desarrollados en este capitulo.

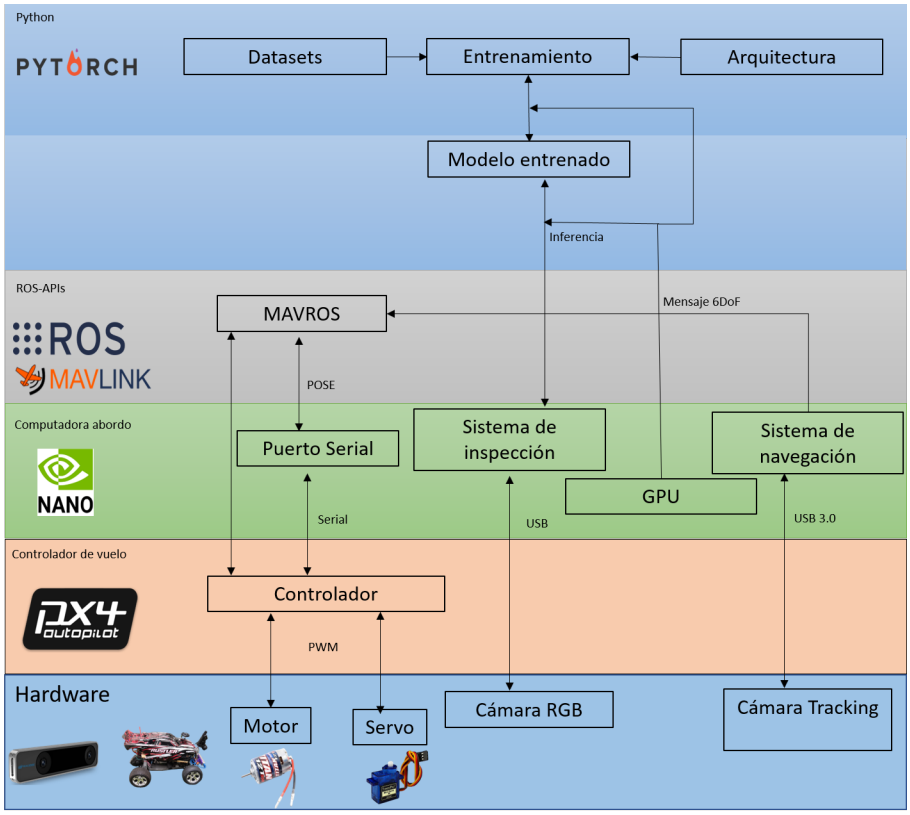


Figura 3.1: Sistemas propuestos en el proyecto

3.1 Diseño e integración de plataforma robótica móvil

La integración de la plataforma robótica móvil es de gran importancia para el proyecto, debido que en ella se implementaran y agregaran todos los componentes del sistema de navegación basado en visión, así como el sistema de detección de fallas en el pavimento además del sistema de potencia del robot. El robot móvil se conforma por un controlador, controladores de motores, motores, sensores, baterías, computadora embebida entre otros.

3.1.1. Diseño de hardware

El diseño de la plataforma a utilizar debe de contemplar los requerimientos para la operación adecuada bajo las condiciones de prueba, considerando el peso y dimensiones de los componentes que integran los sistemas de visión a implementar.

- La plataforma debe tener un espacio suficiente para albergar los componentes electrónicos del sistema.
- La estructura de la plataforma debe tener una rigidez que permita soportar el peso de los componentes electrónicos.
- La estructura debe permitir un grado de movilidad suficiente para la navegación en el espacio de prueba.
- Los motores deben de proporcionar el torque suficiente para vencer la resistencia del sistema.
- Las llantas de la plataforma deben de proporcionar la tracción suficiente para que el sistema se mueva.
- El sistema de alimentación eléctrico debe de proporcionar el voltaje y amperaje de operación de los motores, así como el sistema de comunicación y control.
- El sistema de comunicación de la computadora deberá ser inalámbrico, basado en conectividad Wi-Fi.

3.1.2. Integración de la plataforma robótica móvil

Como base para la plataforma robótica se seleccionó un vehículo móvil de radio-control comercial de la marca Traxxas modelo Rustler XL-5 capaz de albergar los componentes eléctricos del sistema, así como los componentes de comunicación entre otros. La composición a base de nylon del chasis proporciona la rigidez necesaria para soportar el peso de los componentes que integran el sistema. Las dimensiones de la plataforma permiten colocar los componentes electrónicos utilizados en el sistema, el torque proporcionado por el sistema permite una carga máxima de 5 kg además del peso de la estructura, suficientes para la implementación deseada, el sistema diferencial permite una navegación suave y precisa. La figura 3.2 muestra la imagen de la plataforma, la tabla 3.1 contiene las especificaciones de la plataforma brindadas por el fabricante, Traxxas.

3.1.3. Pixhawk 4

Para el control de la plataforma se eligió un controlador de vuelo modelo Pixhawk 4, figura 3.3, de la empresa Holybro, el cual es una vertiente del proyecto Pixhawk basado en open-hardware, el cual busca proveer un estándar en plataformas autopilot de bajo costo, diseñados para propósitos académicos y comunidades de desarrollo. El controlador Pixhawk 4 posee las capacidades de hardware necesarias para la etapa de control del proyecto, el voltaje de operación requerido está dentro del rango

Figura 3.2: Vehículo RC Traxxas Rustler XL-5 (Traxxas, 2020)

Tabla 3.1: Especificaciones Rustler (Traxxas, 2020)

Especificaciones	
Largo	445 mm
Ancho	311 mm
Peso	1.69 Kg
Altura	178 mm
Distancia entre ejes	289 mm
Material de llantas	Caucho
Control electrónico de velocidad	XL-5
Motor	Titan 12T 550
Servomotor de dirección	2056
Tipo de diferencial	Planetario
Velocidad maxima	55 kmh
Bateria	Li-po 3000 mAh 8.4v

de 4.9-5.5 v, con una corriente máxima de 120 A, puede ser alimentado por USB o a través de su placa de regulación. Las interfaces soportadas por el controlador incluyen PWM, necesario para el control del motor y servomotor de la plataforma robótica, además de comunicación serial, necesaria para la interacción con la computadora del proyecto.

3.1.4. Tablilla de administración de energía (PMB)

El pixhawk 4 cuenta con una tablilla de administración de energía (PMB, por sus siglas en inglés) 3.4, la cual alimenta el controlador de vuelo y los actuadores de nuestra plataforma. Para lidiar con el amperaje requerido el PMB cuenta con una capacidad de corriente de 120A, el resto de las especificaciones se pueden visualizar en la tabla 3.2.

Figura 3.3: Pixhawk 4 (Holybro, 2019)

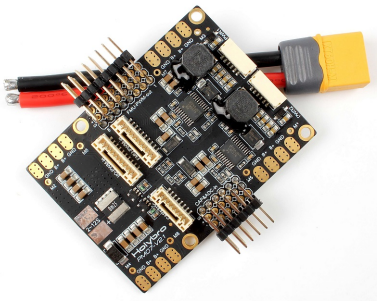


Figura 3.4: PMB (Holybro, 2019)

Tabla 3.2: Especificaciones PMB (Traxxas, 2020)

Especificaciones	
Corriente	120A Max
Corriente de salida UBEC 5V	3A
Voltaje de entrada UBEC	7 51v (2 12s LiPo)
Dimensiones	$68 \times 50 \times 8$ mm
Hoyos de montado	45×45 mm
Peso	36g

3.1.5. Realsense T265

Para el sistema de navegación por visión se optó por utilizar la cámara Realsense T265, figura 3.5, de Intel, la cual posee dos cámaras tipo Fisheye modelo OV9282 con un campo de visión de 163 grados, además de una unidad de procesamiento visual optimizado para algoritmos V-SLAM, mismos que están embebidos en la cámara, la conectividad es mediante USB 3.1, las dimensiones son 108 mm x 24.5 mm x 12.5 mm, los componentes del dispositivo son listados en la tabla 3.3.

3.1.6. Jetson Nano

Para el procesamiento de la información se seleccionó una computadora Jetson Nano de la marca NVIDIA, figura 3.6, perteneciente a la división Jetson la cual está enfocada en aplicaciones de inteligencia artificial y robótica, esta computadora se caracteriza por estar integrada en una sola tablilla, posee unas dimensiones de 69x45 mm. El sistema operativo no está incluido en el sistema, sin embargo, es posible utilizar Linux, específicamente una distribución de Ubuntu ideal para la plataforma, las

Figura 3.5: Realsense T265 (Intel, 2019b)

Tabla 3.3: Componentes Realsense T265 (Intel, 2019b)

Componente	Descripción
BMI055 IMU	Acelerómetro y giroscopio
OV9282	Cámara monocromática
Movidius MA215x	Procesador VPU
Refuerzo	Refuerzo de cubierta, para mantener las imágenes alineadas
Regulador de voltaje	Min 4.5 v Max 5.25 v

características del software libre hacen ideal la implementación de este sistema operativo en el proyecto. En cuanto a las características esta computadora cuenta con un procesador de cuatro núcleos a 1.43 GHz y una unidad cuda para procesamiento de tareas de inteligencia artificial de 128 núcleos, el resto de sus características esta listada en la tabla 3.4

3.1.7. Diseño de piezas impresas 3D

Para la integración de los componentes en la plataforma es necesario el diseño de algunos componentes para su impresión en 3D, el material utilizado para la impresión es ácido poliláctico conocido como PLA, es común su uso en este tipo de aplicaciones debido a sus propiedades y costo. Este termoplástico es soluble en agua y puede ser enjuagado y reutilizado. La aplicación considerada para las piezas no requiere propiedades altas de resistencia al calor ni será sometida a un estrés alto, solo servirán como soportes para la colocación de componentes de bajo peso. El software de diseño utilizado es Solidworks, software de diseño CAD que permite el modelado mecánico de piezas en 2D y 3D.

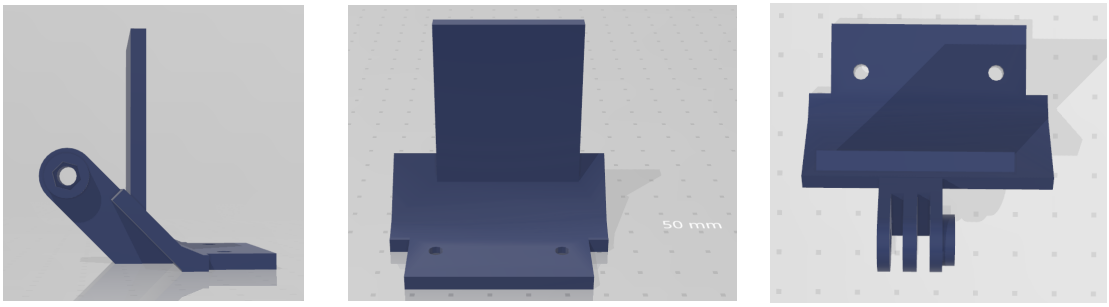
Se requiere un total de tres piezas, un soporte para colocar la cámara T265 de Realsense que será colocado al frente de la plataforma, un soporte para cámara RGB y un soporte que permita montar la cámara T265 de Realsense en el soporte del vehículo.

Figura 3.6: Jetson Nano (Nvidia, 2019)

Tabla 3.4: Especificaciones Jetson Nano (Nvidia, 2019)

Especificaciones	
GPU	Maxwell 128-Núcleos
CPU	ARM A57 Quad-Core 1.43
Memoria	4 GB
Almacenamiento	Micro SD
Conectividad	Ethernet (Inalámbrico no incluido)
Visualización	HDMI
USB	4X USB 3.0
Dimensiones	69*45 mm

En las figura 3.7 se observa el diseño del soporte para la cámara RGB de la plataforma, con sus vistas lateral, frontal y superior se sustituye una pieza de la defensa de la plataforma original, utilizando los mismos tornillos para su sujeción.



(a) Vista lateral **(b) Vista frontal** **(c) Vista superior**

Figura 3.7: Soporte para cámara RGB

En la figura 3.8 se observa el diseño del soporte que será montado en el vehículo, el diseño para la unión de la pieza que va en la cámara T265 de Realsense, es similar al utilizado por la marca GoPro, permitiendo una conexión tipo “Plug and play”.

La figura 3.9 muestra las vistas superior y lateral de la pieza utilizada para sujetar la cámara T265 de Realsense en la plataforma, los tornillos utilizados son M3*5mm, medida utilizada en los orificios

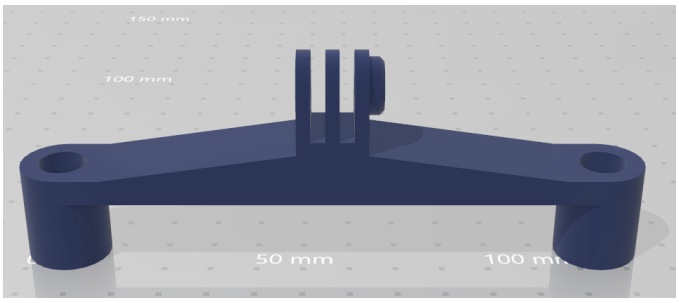
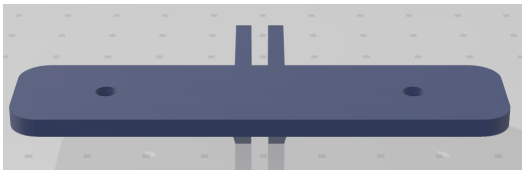
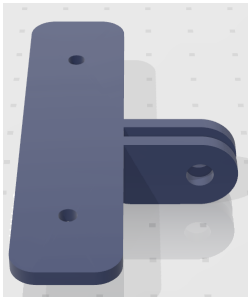


Figura 3.8: Soporte para plataforma (Vista frontal)

originales de la cámara, el conector para la integración en la plataforma es tipo hembra de la forma utilizada en las cámaras de la marca GoPro.



(a) Vista superior



(b) Vista lateral

Figura 3.9: Soporte para cámara T265 Realsense

3.1.8. Plataforma integrada

Finalmente la figura 3.10 muestra la plataforma integrada, con los componentes que integran el vehículo a escala así como su ubicación y presentación. Los componentes del vehículo están colocados en 2 niveles, en el primer nivel se coloco la batería que alimenta el sistema y los actuadores, las computadoras de vuelo y a bordo están colocadas en el segundo nivel. Para proteger los componentes y cableado de la plataforma se opto por utilizar la carcasa del vehículo 3.11.

3.2 Configuración de software

En esta etapa es necesario configurar el ambiente de programación a utilizar en la computadora abordo Jetson Nano.

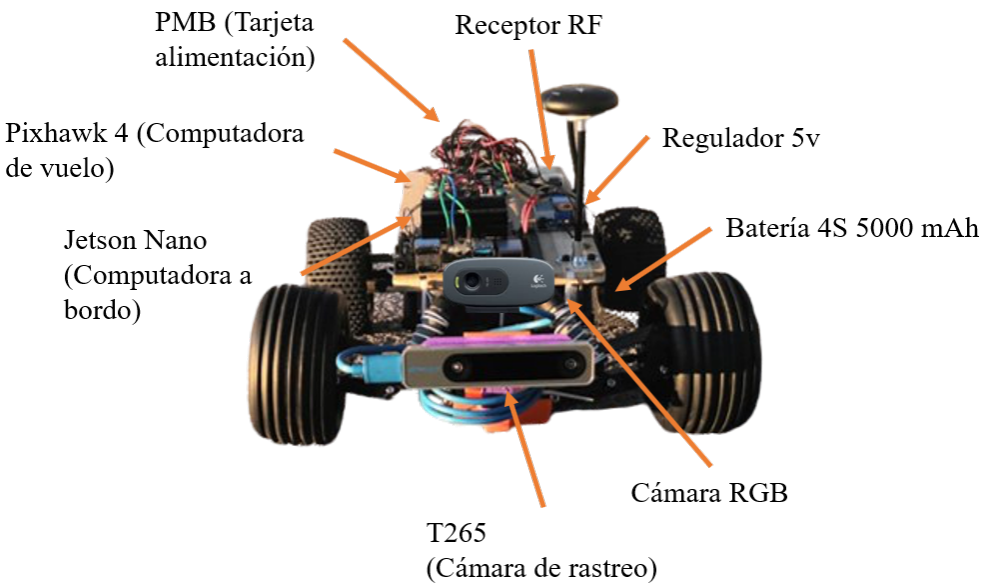


Figura 3.10: Componentes de la plataforma

Figura 3.11: Plataforma con cubierta

3.2.1. Instalación Ubuntu 18.04

El sistema operativo que correrá en el Jetson Nano es la distribución de Ubuntu 18.04 modificada por Nvidia, la cual contiene entornos de desarrollo para inteligencia artificial, conocidos como Jetpack, los pasos para esta instalación están listados a continuación:

1. Descargar la imagen en un ordenador del siguiente link https://developer.nvidia.com/jetson-Nano-sd-card-image.

2. Instalar software para creación de memorias SD de arranque, en este caso, Etcher, disponible en el siguiente enlace: https://www.balena.io/etcher.

3. Instalar imagen en la memoria SD, Figura 3.12.

4. Insertar memoria SD, en el Jetson Nano, es necesario conectar periféricos tales como monitor, teclado y ratón.
5. Conectar dispositivo a fuente de alimentación (5v 4A).
6. Continuar con el proceso de configuración, seleccionar zona horaria, idioma teclado, red wifi y contraseña.
7. El sistema operativo estará configurado para los siguientes pasos.

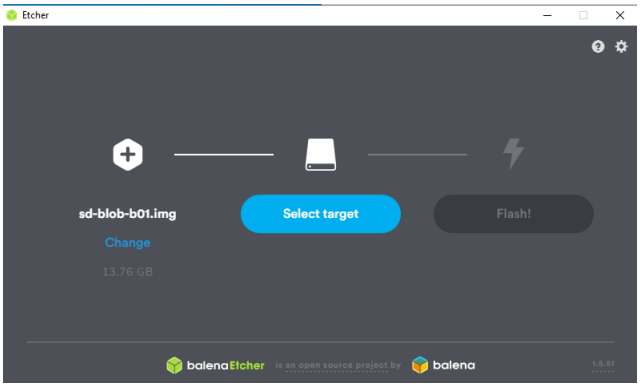


Figura 3.12: Software balena etcher

3.2.2. Robotic Operating System (ROS)

El siguiente paso es instalar nuestro entorno de desarrollo ROS, el cual será utilizado para establecer una comunicación entre bajo y alto nivel, es decir el software de nuestro sistema y el hardware de la plataforma. Las instrucciones de instalación pueden ser encontradas en el siguiente enlace: http://wiki.ros.org/kinetic/Installation.

1. Ubuntu 18.04 solo soporta la versión ROS Melodic.
2. Es necesario actualizar los directorios de Ubuntu para continuar con la instalación mediante el siguiente código:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu \$(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. Configurar llaves de Ubuntu con el siguiente código:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80'
--recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

4. Actualizar índice del repositorio apt:

```
sudo apt-get update
```

5. Proceder con la instalación con el siguiente código:

```
sudo apt-get install ros-melodic-desktop-full
```

6. Después de la instalación es necesario correr el siguiente código:

```
sudo rosdep init
rosdep update
```

7. En este paso se configura el ambiente de trabajo, donde las variables serán adheridas automáticamente a la sesión:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

3.2.3. MAVROS

MAVROS es una librería que permite comunicar dispositivos bajo el protocolo Mavlink con ROS. El protocolo Mavlink es popular en su implementación con drones y rovers. Pixhawk integra este protocolo de comunicación. A continuación, se describe el proceso de instalación en nuestro ambiente de programación:

1. En terminal, escribir el siguiente código:

```
sudo apt-get install ros-kinetic-mavros ros-kinetic-mavros-extras
wget https://raw.githubusercontent.com/mavlink/mavros/master/mavros
/scripts/install_geographiclib_datasets.sh
chmod a+x install_geographiclib_datasets.sh
./install_geographiclib_datasets.sh
```

2. La versión de escritorio requiere instalar el plugin RQT:

```
sudo apt-get install ros-kinetic-rqt
ros-kinetic-rqt-common-plugins ros-kinetic-rqt-robot-plugins
```

3.2.4. Librealsense SDK

La empresa Intel provee una plataforma de desarrollo para sus productos Realsense, misma que está disponible para Ubuntu 18.04, a continuación, se describe el proceso de instalación para la versión 2.0:

1. Con el siguiente código se registra la llave pública del servidor donde están alojados los archivos a descargar:

```
sudo apt-key adv --keyserver keys.gnupg.net
--recv-key F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE ||
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv-key F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE
```

2. Agregar repositorio apt:

```
sudo add-apt-repository
"deb http://realsense-hw-public.s3.amazonaws.com/Debian/apt-repo bionic main"
-u
```

3. Instalar librerias:

```
sudo apt-get install librealsense2-dkms
sudo apt-get install librealsense2-utils
```

4. Verificar instalación, el siguiente código inicia el software de visualización:

```
realsense-viewer
```

5. Después de correr el código deberá de aparecer una ventana igual a la figura 3.13.

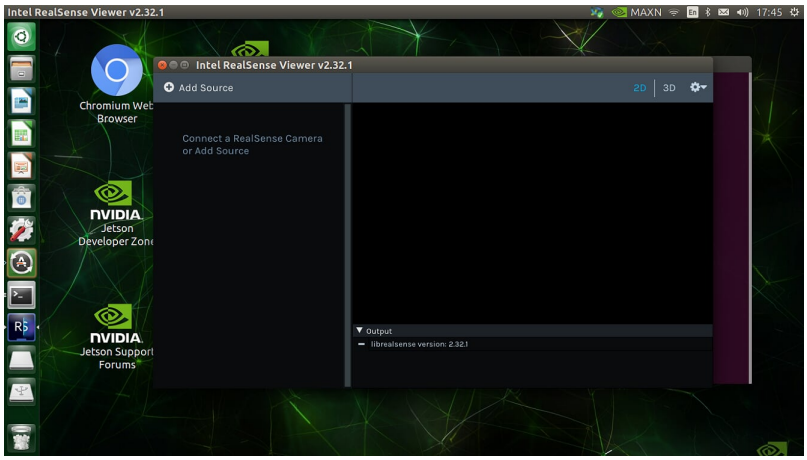


Figura 3.13: Intel RealSense Viewer

3.2.5. Python 3

El lenguaje de programación de este proyecto está basado principalmente en Python, por lo que es necesario instalar este software en nuestro entorno, siguiendo las instrucciones de la web oficial, Python puede ser instalado con el siguiente código:

```
sudo apt install python3
```

PIP (Gestor de librerías)

Además, es necesario un administrador de librerías de python, en este caso PIP. El cual es instalado con el siguiente comando:

```
sudo apt install python3-pip
```

Pyrealsense2

Para establecer comunicación entre Python y la librería de Realsense utilizamos la librería Pyrealsense.

```
Pip3 install pyrealsense2 --user
```

RealSense-ROS

Para establecer comunicación entre el entorno de desarrollo Intel Realsense y ROS utilizamos el envoltorio de funciones proveído por Intel, RealSense-ROS. Sin embargo el proceso de instalación para el Jetson Nano, no es directo, algunas modificaciones son necesarias, la pagina web https://github.com/JetsonHacksNano/installRealSenseROS provee un script para esta instalación:

```
git clone https://github.com/JetsonHacksNano/installRealSenseROS
cd installRealSenseROS
./installRealSenseROS catkin_ws
```

3.3 Instrumentación electrónica

La integración electrónica de los elementos de la plataforma es llevada en 2 etapas diferentes, la etapa de potencia y la etapa de comunicación.

3.3.1. Etapa de potencia

El circuito es alimentado por una batería tipo lipo de 7.4 v. Por su parte el Pixhawk 4 recibe energía directamente de la batería debido a que cuenta con un regulador interno, por otra lado, el Jetson Nano no posee un regulador interno, por lo cual un regulador de 5 V y 4 A es utilizado para alimentarlo, utilizando los puertos 5V y GND de las entradas y salidas del Jetson Nano (GPIO) apéndice A.1.1 figura A.1. El controlador de los motores, el motor DC y el servomotor son alimentados directamente por la batería. Las señales de control del controlador (ESC) son conectadas a su motor correspondiente. El Pixhawk 4 a través de los puertos 1 y 3 envía las señales de control al controlador de los motores. La figura 3.14 muestra una representación gráfica de dichas conexiones, la figura 3.15 muestra el esquemático del circuito eléctrico.

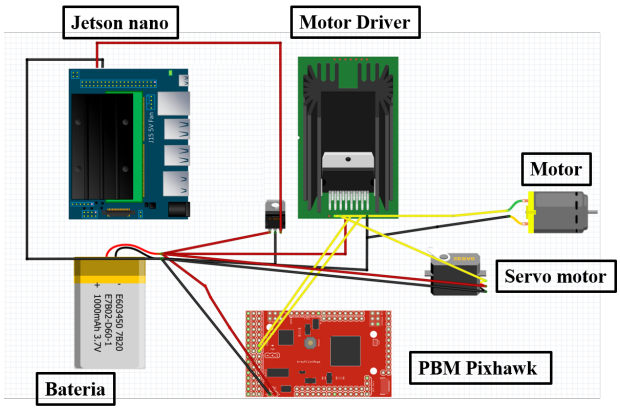


Figura 3.14: Diagrama de conexión etapa de potencia.

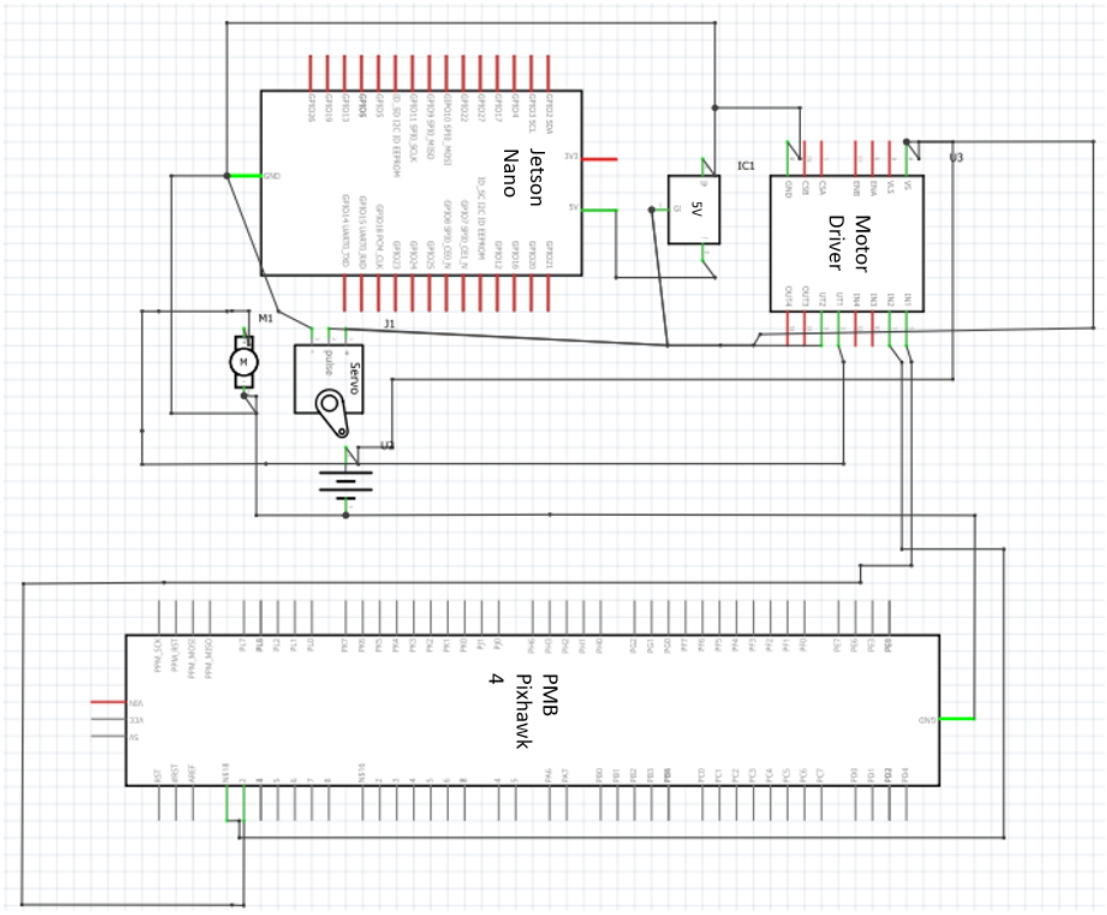


Figura 3.15: Esquema de diagrama eléctrico

3.3.2. Etapa de comunicación

En esta etapa se conectan los periféricos utilizados a la computadora. La computadora Jetson Nano cuenta con 4 puertos USB. La cámara Realsense T265 es conectada mediante USB 3.0, mientras que la cámara RGB es conectada utilizando USB 2.0. El Pixhawk 4 permite comunicación serial mediante dos puertos de comunicación, por su parte el Jetson Nano cuenta con dos puertos para comunicación Serial, sin embargo, utilizando un cable TTL USB es posible realizar la conexión a través de los puertos USB del dispositivo Jetson Nano, la tabla 3.5 muestra la distribución de pines de los puertos de comunicación del Pixhawk 4, se debe realizar la conexión correspondiente con el cable TTL-USB figura 3.17. El voltaje lógico de comunicación es de 3.3 V tanto en el Jetson Nano como el Pixhaw 4, por lo tanto la comunicación por el puerto serie será directa. La conexión inalámbrica al Jetson Nano sucede a través del protocolo Wi-Fi. A sí mismo la conexión del Pixhawk 4 con la estación de control terrestre se da de manera inalámbrica utilizando un dispositivo Bluetooth, la figura 3.16 muestra el diagrama de conexión.

Figura 3.16: Diagrama de protocolos de comunicación.

Tabla 3.5: Distribución de pines de los puertos de comunicación Pixhawk 4

TELEM 1, 2 Pixhawk 4		
Pin	Señal	Voltaje
1 (Rojo)	Vcc	+5V
2 (Negro)	TX (Salida)	+3.3V
3 (Negro)	RX (Entrada)	+3.3V
4 (Negro)	CTS (Entrada)	+3.3V
5 (Negro)	RTS (Salida)	3.3V
6 (Negro)	GND	GND

Figura 3.17: Cable USB a serial TTL

3.4 Desarrollo de sistema de navegación basado en visión.

En esta etapa del proyecto se desarrolló un algoritmo en ROS-Python para convertir la información visual de la cámara T265 en coordenadas y orientaciones del sistema y después en información del desplazamiento en el espacio del vehículo para que a su vez que pueda ser interpretadas por el controlador de vuelo (Pixhawk 4).

3.4.1. Dibujo cinemático

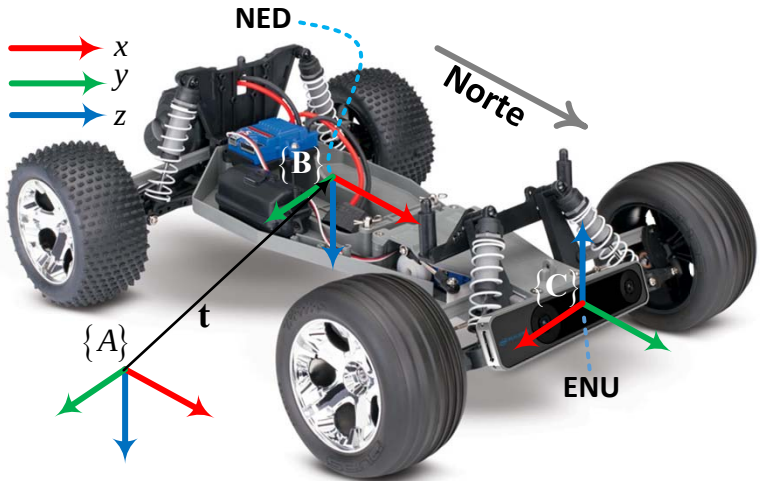


Figura 3.18: Diagrama cinemático

3.4.2. Marco de referencia

Nuestro sensor de visión trabaja bajo su propio marco de referencia, por lo cual los desplazamientos que calculemos deberán tener en cuenta esto, por otra parte ROS utiliza el marco de referencia ENU, mientras que el controlador Pixhawk al ser enfocado en drones trabaja bajo el marco de referencia NED.

Por lo tanto, es necesario transformar el marco de referencia de la librería de ROS de la cámara $\{C\}$ al marco de referencia rígido $\{B\}$ figura 3.18, una rotación de 180° sobre el eje X del marco $\{C\}$

alineará el eje Z con el eje Z del marco de referencia $\{\mathrm{B}\}$, una rotación -90° sobre el eje Z alineará el eje X y el eje Y del marco de referencia $\{\mathrm{B}\}$. La matriz de rotación ${}^{B}\mathbf{R}_{C}$ definida en la ecuación 3.1 nos ayuda a realizar esta transformación del marco de referencia $\{\mathrm{C}\}$ al marco de referencia $\{\mathrm{B}\}$.

$$ {}^{B}\mathbf{R}_{C} = \begin{bmatrix} \cos -90^\circ & \sin -90^\circ & 0 \\ -\sin -90^\circ & \cos -90^\circ & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos 180^\circ & \sin 180^\circ \\ 0 & -\sin 180^\circ & \cos 180^\circ \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{3.1} $$

3.4.3. Orientación

Para representar la orientación de nuestro robot en tres dimensiones respecto a un marco de referencia utilizamos los ángulos RPY, el cual se puede representar en un vector de características $\boldsymbol{o} \in \mathbb{R}^{3\times 1}$, visible en la ecuación 3.2.

$$ \mathbf{o}^{\mathsf{T}} = \begin{bmatrix} o_x & o_y & o_z \end{bmatrix} \tag{3.2} $$

3.4.4. Pose en robótica

En robótica y visión artificial, determinar la posición y orientación de un robot es comúnmente llamada pose, por lo tanto, un vector pose, posee la información del vector posición y el vector orientación. En un robot móvil terrestre lo orientación en o_x y o_y no es modificada a través del tiempo en una superficie plana. Por lo tanto el vector pose ($\mathbf{w}$), se compone de la unión de los vectores de posición 2.1 y orientación 3.2, ecuación 3.3.

$$ \mathbf{w}^{\mathsf{T}} = \begin{bmatrix} p_x & p_y & p_z & o_x & o_y & o_z \end{bmatrix} \tag{3.3} $$

3.4.5. Determinación de nueva posición

Para determinar el desplazamiento de nuestro robot respecto al marco de referencia fijo $\{\mathrm{A}\}$, es necesario construir la matriz de transformación homogénea, debido a que el marco de referencia $\{\mathrm{B}\}$ posee la misma orientación que el marco de referencia $\{\mathrm{A}\}$, la matriz de rotación será la matriz identidad, por lo tanto la matriz de transformación homogénea del sistema queda definida en la ecuación 3.4.

$$ {}^{B}\mathbf{T}_{A} = \begin{bmatrix} 1 & 0 & 0 & \mathbf{t}_x \\ 0 & 1 & 0 & \mathbf{t}_y \\ 0 & 0 & 1 & \mathbf{t}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4} $$

Donde el desplazamiento entre la posición anterior y la actual esta representado en el vector $\boldsymbol{t}$.

3.4.6. Algoritmo de conversión

La cámara T265 trasmite un vector pose de tipo entero con 6 elementos, mediante la comunicación USB, para poder utilizar esta información es necesario convertir la información al protocolo de comunicación ROS, posteriormente una vez calculada la nueva posición deberemos enviar el vector pose al controlador de vuelo, por lo tanto, una vez más debe ser transformada la información al protocolo Mavlink. Este flujo de datos esta representado en la figura 3.19, donde el elemento 1 es la adquisición de datos en tiempo real, mientras que los elementos 2, 3 y 4 son nodos de ROS, el ultimo elemento es la recepción del mensaje por el Pixhawk 4.

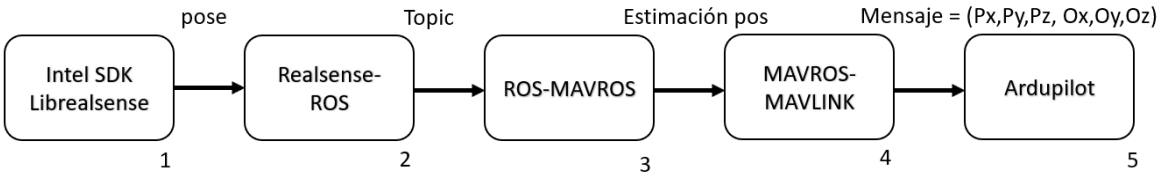


Figura 3.19: Flujo de datos

Para calcular Δ de desplazamiento en los tres ejes se emplea el siguiente código:

```
delta_tras = [data.tras.x - prev_data.tras.x, data.tras.y -
prev_data.tras.y, data.tras.z - prev_data.tras.z]
```

3.4.7. Establecimiento del origen

Al prescindir del uso de GPS, el sistema no puede determinar por si mismo su posición global, es necesario determinar de manera manual el origen en el cual se iniciara a registrar la posición, para esto se hace uso de tres variables globales de tipo entero:

```
lat = 31743254
lon = -106430562
alt = 1120
```

Los valores indicados representan la ubicación del laboratorio de automatización de la universidad Autónoma de Ciudad Juárez en coordenadas geodésicas, esto de acuerdo a Google Maps (Google maps, 2020) figura 3.20, es necesario utilizar los valores con unidades de tipo entero, es decir sin utilizar el punto decimal.

3.4.8. Compensación por software

Debido a las limitaciones físicas de nuestra plataforma, la cámara T265 no puede ser colocada en el centroide figura 3.21.

Esta diferencia puede ser compensada mediante software, con un script de python podemos ajustar la diferencia para simular que la cámara esta colocada en el centroide de la plataforma, como en

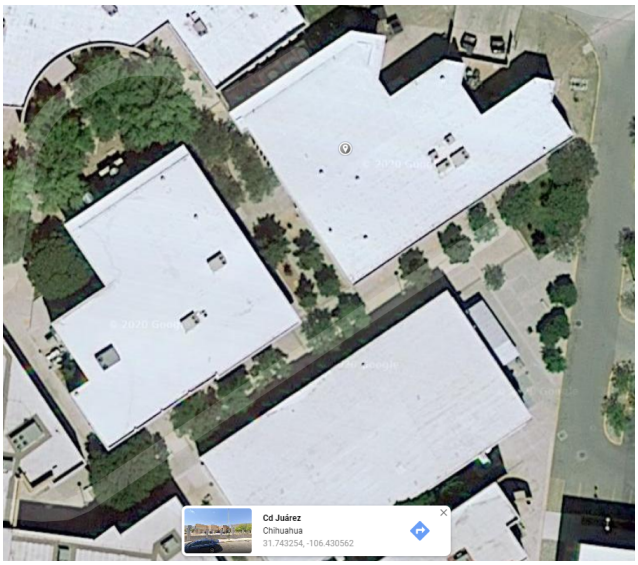


Figura 3.20: Vista aérea del laboratorio de automatización UACJ (Google, 2020)

nuestra implementación solamente esta desfasada en el eje x, solo esta variable será modificada, como resultado tendremos una posición de cámara simulada figura 3.22.

```
comp_x = 0.225  # En metros (m)
comp_y = 0  # En metros (m)
comp_z = 0  # En metros (m)
```

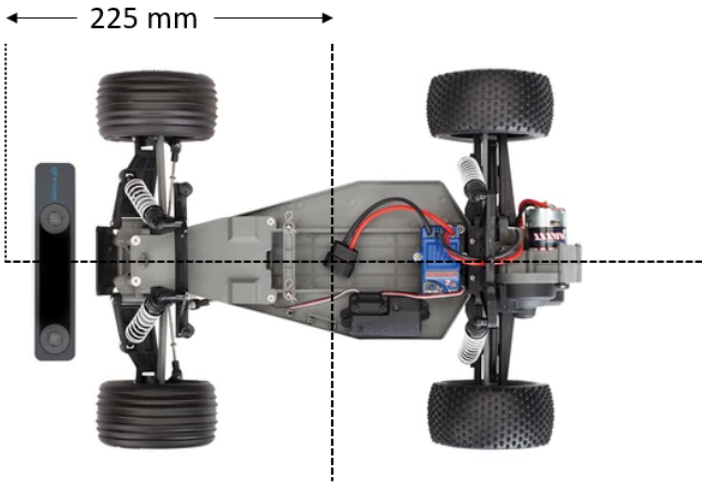


Figura 3.21: Distancia entre posición de la cámara y centroide de la plataforma

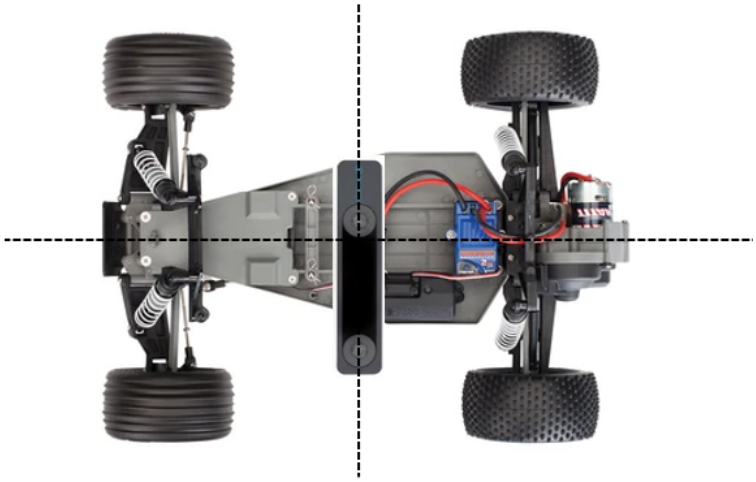


Figura 3.22: Compensación por software

3.4.9. Alineación al norte

Al iniciar el algoritmo, este asume que la orientación de la plataforma esta hacia el norte, por lo cual es necesario colocar de manera manual la plataforma con el frente del vehículo hacia el norte figura 3.23, esto puede ser logrado utilizando la aplicación brújula de un dispositivo móvil figura 3.24.

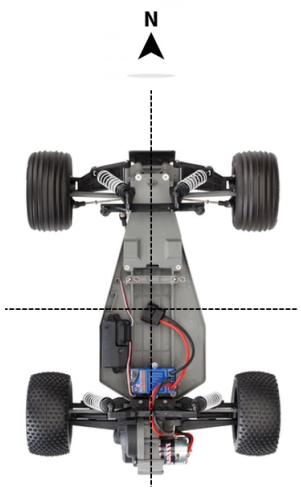


Figura 3.23: Alineación al norte

Figura 3.24: Aplicación brújula

3.4.10. Parámetros de Mission Planner

Para poder recibir el mensaje de posición creado en el algoritmo, es necesario deshabilitar el uso del GPS en el pixhawk, además de habilitar la comunicación serial por la cual la computadora Jetson Nano enviará el mensaje. Estos parámetros se muestran el la tabla 3.6.

Tabla 3.6: Parámetros modificados en Mission Planner

Especificaciones	
AHRS_EKF_TYPE	2
EK2_ENABLE	1
EK3_ENABLE	0
EK2_GPS_TYPE	3
EK2_POSNE_M_NSE	0.1
EK2_VELD_M_NSE	0.1
EK2_VELNE_M_NSE	0.1
BRD_RTC_TYPES	2
GPS_TYPE	0
COMPASS_USE	0
COMPASS_USE2	0
COMPASS_USE3	0
SERIAL5_BAUD	921
SERIAL5_PROTOCOL	1
SYSID_MYGCS	1

3.4.11. Conexión Serial Jetson Nano-Pixhawk 4

Como se estableció la comunicacion entre el Jetson Nano y Pixhawk 4 se realiza mediante comunicacion serial, el cable configurado en el capitulo 2 convierte TTL a comunicacion USB, es necesario conocer el puerto asignado en el Jetson Nano, esto se logra con el siguiente comando:

```
for sysdevpath in $(find /sys/bus/usb/devices/usb*/ -name dev); do
    (
        syspath="${sysdevpath\%/dev}"
        devname="$(udevadm info -q name -p $syspath)"
        [[ "$devname" == "bus/"* ]] && continue
        eval "$(udevadm info -q property --export -p $syspath)"
        [[ -z "$ID_SERIAL" ]] && continue
        echo "/dev/$devname - $ID_SERIAL"
    )
done
```

En nuestro caso el puerto utilizado es:

```
ttyUSB0
```

En la tabla 3.6 el parámetro SERIAL5_BAUD=921 representa una velocidad de 921600 baudios por lo que deberemos configurar esta velocidad en nuestro código de inicio, dicho código esta descrito a continuación:

```
roslaunch mavros apm.launch fcu_url:=/dev/ttyUSB0:921600
```

Si se niega la conexión debido a permisos de escritura, es necesario modificar los parámetros del puerto, esto se logra mediante el comando:

```
sudo chmod 666 /dev/ttyUSB0
```

Lograda la conexión deberemos de recibir la retroalimentación de la terminal que se estableció la conexión con éxito, figura 3.25

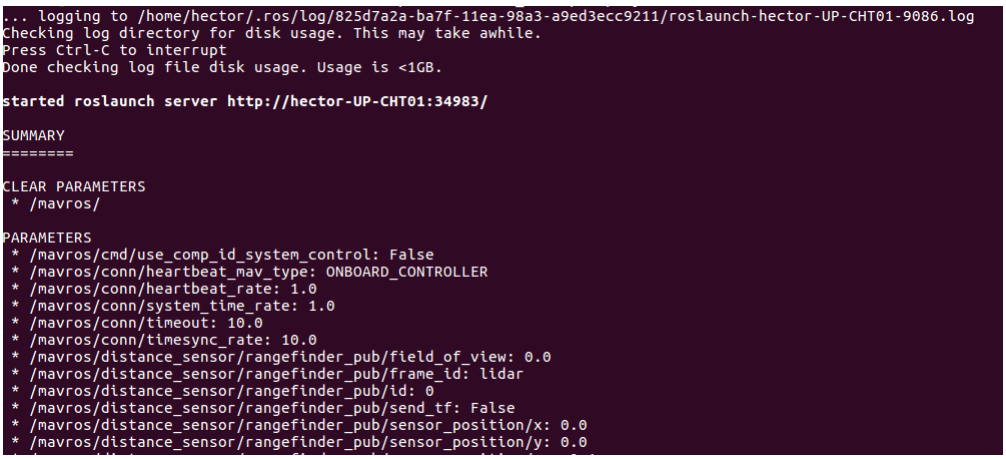


Figura 3.25: Conexión Jetson-Pixhawk4

3.4.12. Inicialización de sistema de navegación

Establecidos los parámetros de conexión, procedemos a iniciar el sistema de navegación en ROS, el código completo es incluido en el apéndice A.2.4.

- Nodo T265, este nodo es proporcionado por Intel, apéndice A.2.2, nodo que permite inicializar la cámara T265 y la captura de datos.

  ```
  roslaunch realsense2_camera rs_t265.launch
  ```

- Nodo de odometría y conversión ROS-Mavlink.

  ```
  roslaunch t265_to_mavros camarat265_pose_to_mavros.launch
  ```

- Nodo de visualización en terminal, con este nodo podremos observar la salida de nuestro nodo de conversión a una velocidad de 30 Hz.

```
rostopic hz /mavros/t265_pose/
```

3.4.13. Costo computacional - Navegación

Ejecutar el algoritmo en el Jetson Nano supone una cantidad de instrucciones y cálculos por segundo para establecer el envió de los datos de odometría en tiempo real, esto se ve reflejado en el consumo de recursos de la figura 3.26, donde los cuatro núcleos del jetson Nano son utilizados hasta un 55 % de su capacidad, el procesamiento de imágenes es realizado directamente en la cámara T265, por lo cual el consumo del GPU se mantiene en 5 %, con picos máximos de 15 %.

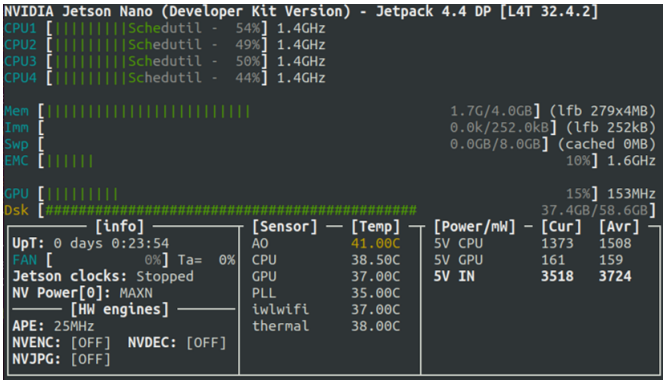


Figura 3.26: Consumo de recursos-Algoritmo navegación

4. Sistema de inspección inteligente de pavimento

4.1 Entrenamiento de red neuronal

Para detectar fallas en el pavimento es necesario realizar el procesamiento de las imágenes adquiridas por nuestro sensor, en esta caso la cámara RGB, un enfoque clásico de procesamiento de imágenes no puede resolver este problema debido a las características variadas de una falla o agrietamiento en el pavimento o asfalto, este problema se representa en la figura 4.1, donde están presentes círculos y triángulos de diferente color, no es posible dividir los círculos utilizando una línea recta (regla), no se puede decir que las figuras de color negro son círculos, ni que los cuadrantes de la derecha son círculos, por lo cual no hay algoritmo capaz de converger en esta condición. Sin embargo, utilizando el enfoque de inteligencia artificial, podríamos establecer un modelo que nos permita inferir en cuales cuadrantes se encuentran los círculos, figura 4.2.

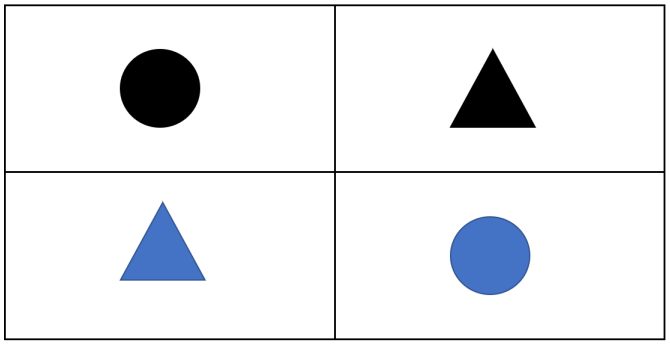


Figura 4.1: Problema no lineal

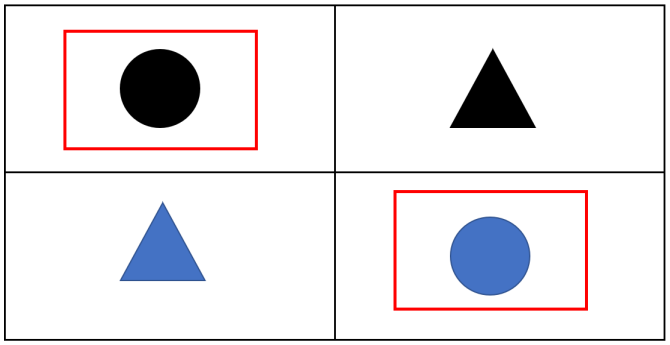


Figura 4.2: Solución utilizando IA

4.2 Software

4.2.1. Jetson-Inference

Para poder utilizar redes neuronales en el CUDA del Jetson nano, es necesario contar con las librerías publicadas por NVIDIA, el proceso de instalación es el siguiente.

1. Instalar Cmake:

```
sudo apt-get update
sudo apt-get install git cmake
```

2. Descargar repositorio:

```
git clone https://github.com/dusty-nv/Jetson-inference
cd Jetson-inference
git submodule update --init
```

3. Instalar librerías de desarrollo de Python.

```
sudo apt-get install libpython3-dev python3-numpy
```

4. Configurar utilizando Cmake:

```
cd Jetson-inference
mkdir build
cd build
```

4.2.2. PyTorch

Uno de los entornos de desarrollo de inteligencia artificial utilizados en este proyecto es PyTorch, para instalar en el Jetson nano es necesario utilizar los siguientes comandos en terminal:

```
 cd Jetson-inference/build
./install-pytorch.sh
```

5. Compilar proyecto utilizando Cmake, para crear las librerías y las extensiones de Python:

```
cd Jetson-inference/build
make
sudo make install
sudo ldconfig
```

4.3 Creación de base de datos

En esta etapa se define la base de datos que se utilizará para entrenar nuestra red neuronal, existen varios factores al tener en consideración en la construcción de nuestra base de datos, la variación entre las dimensiones, colores y ángulos de las grietas visibles en las imágenes permitirá generar un modelo más robusto. Este, al ser un problema de clasificación binario, requiere un conjunto de datos para cada una de las dos variables, fallas y no fallas, es decir, conjuntos de imágenes donde este presente la falla y un conjunto de imágenes donde no se aprecien fallas. El tamaño de la base de datos generado es definido por las siguientes consideraciones.

- Capacidad de computo para el entrenamiento.
- Tiempo de entrenamiento.
- Sobre ajuste (Overfitting).

Se eligió construir un conjunto de imágenes para este proyecto utilizando tres subconjuntos de datos.

- Conjunto propio, con imágenes de la localidad.
- Conjunto SDNET2018 (Dorafshan, Thomas, y Maguire, 2018), sub sección de pavimentos.
- Conjunto de imágenes para clasificación de grietas (Zhang, Yang, Daniel Zhang, y Zhu, 2016).

Se seleccionaron imágenes aleatorias para cada una de las variables hasta llegar a 5500 imágenes para cada clase, 10000 en total, de las cuales 5000 imágenes de cada clase pertenecerán al entrenamiento y 500 a la validación.

4.4 Definición de arquitectura

Al trabajar con redes neuronales para problemas de clasificación, se puede utilizar redes neuronales (NN), sin embargo, al utilizar imágenes como variable de entrada resulta mas conveniente utilizar redes neuronales convolucionales (CNN), las imágenes en su forma de representación matricial no contienen información explicita de las características que componen una grieta en pavimento, solo contienen información del color de cada píxel que compone la imagen, para extraer las características relevantes o vector de características, se emplean operaciones convolucionales, utilizando de una serie de filtros aplicados a la imagen de entrada. La figura 4.3 presenta una red neuronal convolucional genérica para resolver el problema de clasificación binaria, con una imagen como entrada, filtro de suavizado en la etapa convolucional, etapa de max pooling, capas totalmente conectadas y una salida binaria. El número de capas que posee la red estará relacionada con el poder computacional donde se realice el entrenamiento, un número elevado de capas aumentará el número de operaciones a realizar.

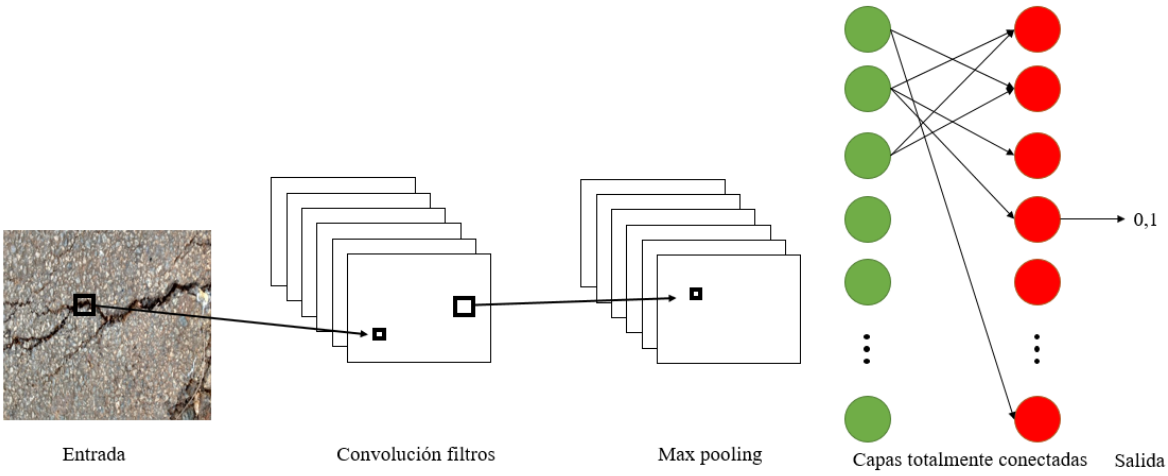


Figura 4.3: Red neuronal convolucional genérica

El modelo Resnet es un modelo pre-entrenado para clasificar objetos de 100 clases diferentes, ganador del concurso ImageNet en 2015 (Kaiming y cols., 2016), originalmente con 100 capas, utilizar un modelo pre-entrenado permite transferir el reconocimiento de características como bordes, sombras, cambios de color, que fueron aprendidos en el entrenamiento original a pesar de que las clases originales no comprenden nuestro problema de estudio, a la vez reduce el tiempo de entrenamiento cuando no se está utilizando hardware especializado. Resnet18 es una implementación de 18 capas lo cual permite que sea lo suficientemente ligera para correr en tiempo real en el Jetson nano. La estructura de esta red esta compuesta por 5 capas ocultas, donde en cada capa se aplicaran filtros convolucionales que extraerán características de cada una de las entradas, el paso (Stride) de cada filtro aplicado será de dos píxeles, posteriormente serán sometidos a un filtro max pool de 3x3 que incrementará las características obtenidas disminuyendo las dimensiones de la matriz. La tabla 4.1 muestra los parámetros de cada capa que componen esta estructura.

Tabla 4.1: Arquitectura Resnet 18 (Kaiming y cols., 2016)

Configuración de red neuronal		
Nombre capa	Tamaño de salida	18-capas
conv 1	112×112	7×7, 64, stride 2
conv 2	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 3 \times 3 & 64 \\ 3 \times 3 & 64 \end{bmatrix} \times 2$
conv 3	28×28	$\begin{bmatrix} 3 \times 3 & 128 \\ 3 \times 3 & 128 \end{bmatrix} \times 2$
conv 4	14×14	$\begin{bmatrix} 3 \times 3 & 256 \\ 3 \times 3 & 256 \end{bmatrix} \times 2$
conv 5	7×7	$\begin{bmatrix} 3 \times 3 & 512 \\ 3 \times 3 & 512 \end{bmatrix} \times 2$
Salida	1×1	average pool, 1000, softmax
FLOPs*		$1.8x10^9$

*Operaciones de punto flotante por segundo.

Del artículo de Kaiming (Kaiming y cols., 2016) se extrajo el bloque que integra el modelo, figura 4.4. Los bloques convolucionales que componen la red realizarán una serie de cálculos, el primero de ellos el convolucional bidimensional donde se calcula el producto tensorial 4.1, la entrada x_0 es de dimensiones (224, 224, 3) y **w** representa la capa de pesos (Capa de convolución).

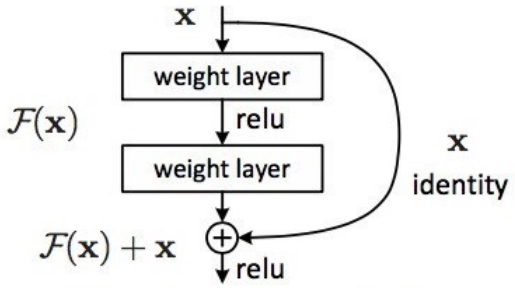


Figura 4.4: Esquema lógico del bloque residual (Kaiming y cols., 2016)

$$y_1(k) = \mathbf{w}_1^k \otimes x_1 \tag{4.1}$$

para $k = 64$, que es el número de filtros aplicados en los bloques de convolución.

Posteriormente se estima la media del lote mediante la ecuación 4.2.

$$\mu^b = \overline{y_1{}^b} \tag{4.2}$$

Donde $y_1{}^b = y_{1,1}{}^b, y_{1,2}{}^b, \ldots, y_{1,8}{}^b$.

Mediante la ecuación 4.3 se calcula la desviación estándar del lote.

$$\sigma^b = \sqrt{\text{Var}[y_1{}^b] + \varepsilon} \tag{4.3}$$

donde ε es una constante pequeña que evita que $\sigma = 0$.

El siguiente cálculo es normalizar respecto al lote utilizando la ecuación 4.4

$$\hat{y}_1 = \frac{y_1 - \mu^b}{\sigma^b} \tag{4.4}$$

Siendo un re-escalamiento con el cual se logra que la media de y_1 sea igual a cero, mientras que la desviación estándar sea igual a uno. El bloque continua aplicando la función de activación de la ecuación 4.5, en este caso dicha función de activación es la función ReLu

$$y_1 = \phi \tilde{y}_1 \tag{4.5}$$

Con esto, las transformaciones realizadas por las primeras capas del bloque convolucional se pueden definir como la función compuesta de F_1 y F_2 ecuación 4.6

$$F := F_2 \circ F_1 \therefore x_2 = F(x_1) \tag{4.6}$$

4.5 Función de activación

La función de activación determina si una neurona se activara o no, es decir si el resultado obtenido en su nodo será enviado a otras neuronas, la función de activación utilizada en este proyecto es un rectificador conocido como unidad lineal rectificada (ReLu por sus siglas en inglés). Este tipo de funciones de activación ofrece mejores resultados de aprendizaje para redes neuronales convolucionales profundas. Este rectificador es definido matemáticamente en la ecuación 4.7, donde x es la entrada de la neurona. Todos los valores menores de cero serán establecidos a cero, mientras que los valores mayores serán enviados a la siguiente neurona sin ser modificados, este comportamiento es visible en la figura 4.5.

$$f(x) = \text{máx}(0, x) \tag{4.7}$$

4.6 Parámetros de entrenamiento

En esta sección se determinan los parámetros de entrenamiento del modelo, el tamaño de las imágenes de entrada será re-escalado a un valor establecido en la sección anterior de 224x224 píxeles, el número de iteraciones donde se analiza el conjunto de datos completo será de 100, el tamaño de lote de cada paso será de $n = 8$, la velocidad de aprendizaje estará establecido en 0.1, y para hacer uso del GPU disponible en el Jetson nano, será habilitado mediante la variable GPU=1, estos parámetros son mostrados en 4.2.

Construir el modelo y parámetros es logrado integrando cada bloque en código python, el primer

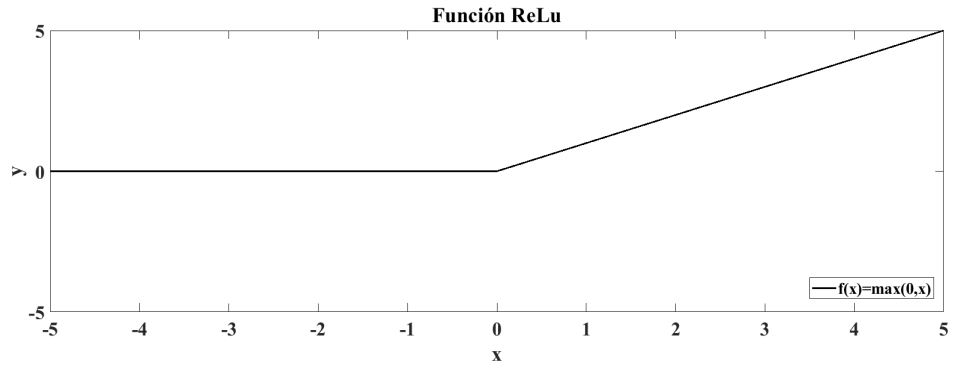


Figura 4.5: Función ReLu

Tabla 4.2: Parámetros de entrenamiento

Parámetro	Valor
Resolución	224x224 píxeles
Iteraciones (Epoch)	100
Tamaño de Lote (Batch size)	8
Velocidad de aprendizaje (Learning rate)	0.1
GPU	1

elemento de dichas capas es visto en el siguiente código:

```
(conv 1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
  track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
  dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3),
      stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1,
      affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
```

El código completo puede ser visto en el apéndice A.2.3. Una vez definidos los parámetros y capas del modelo podemos obtener el resumen de nuestro modelo a entrenar utilizando el siguiente código.

```
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from torchvision import models
from torchsummary import summary

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


resnet18 = models.resnet18().to(device)
summary(resnet18, (3,224, 224))
```

El código anterior produce la tabla 4.3, donde podemos observar el tipo de capa, la forma de salida y el número de parámetros que componen cada capa, el modelo a entrenar posee 11689512 parámetros totales de los cuales el 100 % es entrenable, el tamaño de los archivos de entrada es de 0.57 MB y el de los parámetros 44.59 MB, en total se estima que el peso del modelo sea de 107.96 MB.

Tabla 4.3: Resumen del modelo

Tipo de capa	Forma de salida	No. de Parámetros
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 128, 28, 28]	73,728
BatchNorm2d-20	[-1, 128, 28, 28]	256
ReLU-21	[-1, 128, 28, 28]	0

Continua en la siguiente pagina

Tabla 4.3 – *Continua de la pagina anterior*

Tipo de capa	Forma de salida	No. de Parámetros
Conv2d-22	[-1, 128, 28, 28]	147,456
BatchNorm2d-23	[-1, 128, 28, 28]	256
Conv2d-24	[-1, 128, 28, 28]	8,192
BatchNorm2d-25	[-1, 128, 28, 28]	256
ReLU-26	[-1, 128, 28, 28]	0
BasicBlock-27	[-1, 128, 28, 28]	0
Conv2d-28	[-1, 128, 28, 28]	147,456
BatchNorm2d-29	[-1, 128, 28, 28]	256
ReLU-30	[-1, 128, 28, 28]	0
Conv2d-31	[-1, 128, 28, 28]	147,456
BatchNorm2d-32	[-1, 128, 28, 28]	256
ReLU-33	[-1, 128, 28, 28]	0
BasicBlock-34	[-1, 128, 28, 28]	0
Conv2d-35	[-1, 256, 14, 14]	294,912
BatchNorm2d-36	[-1, 256, 14, 14]	512
ReLU-37	[-1, 256, 14, 14]	0
Conv2d-38	[-1, 256, 14, 14]	589,824
BatchNorm2d-39	[-1, 256, 14, 14]	512
Conv2d-40	[-1, 256, 14, 14]	32,768
BatchNorm2d-41	[-1, 256, 14, 14]	512
ReLU-42	[-1, 256, 14, 14]	0
BasicBlock-43	[-1, 256, 14, 14]	0
Conv2d-44	[-1, 256, 14, 14]	589,824
BatchNorm2d-45	[-1, 256, 14, 14]	512
ReLU-46	[-1, 256, 14, 14]	0
Conv2d-47	[-1, 256, 14, 14]	589,824
BatchNorm2d-48	[-1, 256, 14, 14]	512
ReLU-49	[-1, 256, 14, 14]	0
BasicBlock-50	[-1, 256, 14, 14]	0
Conv2d-51	[-1, 512, 7, 7]	1,179,648
BatchNorm2d-52	[-1, 512, 7, 7]	1,024
ReLU-53	[-1, 512, 7, 7]	0
Conv2d-54	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-55	[-1, 512, 7, 7]	1,024
Conv2d-56	[-1, 512, 7, 7]	131,072
BatchNorm2d-57	[-1, 512, 7, 7]	1,024
ReLU-58	[-1, 512, 7, 7]	0

Continua en la siguiente pagina

Tabla 4.3 – *Continua de la pagina anterior*

Tipo de capa	Forma de salida	No. de Parámetros
BasicBlock-59	[-1, 512, 7, 7]	0
Conv2d-60	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-61	[-1, 512, 7, 7]	1,024
ReLU-62	[-1, 512, 7, 7]	0
Conv2d-63	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-64	[-1, 512, 7, 7]	1,024
ReLU-65	[-1, 512, 7, 7]	0
BasicBlock-66	[-1, 512, 7, 7]	0
AdaptiveAvgPool2d-67	[-1, 512, 1, 1]	0
Linear-68	[-1, 1000]	513,000

Con el modelo configurado se colocan las imágenes que formaran parte del entrenamiento bajo la siguiente estructura:

```
/home/Jetson/datasets/asphalt/train/crack
/home/Jetson/datasets/asphalt/train/nocrack
/home/Jetson/datasets/asphalt/test/crack
/home/Jetson/datasets/asphalt/test/nocrack
/home/Jetson/datasets/asphalt/labels.txt
```

La carpeta “train” posee las imágenes seleccionadas para el entrenamiento dividido en dos carpetas, “crack” donde se colocan las imágenes que presentan una falla en el pavimento y “nocrack” donde se colocan las imágenes que no presentan fallas, de manera similar se colocan dos carpetas dentro de la carpeta “test”donde no se incluirán imágenes que forman parte del entrenamiento bajo las dos variables. El archivo “labels.txt” contiene las etiquetas de las variables del modelo, en este caso son dos, “crack” y “nocrack”. La cantidad de imágenes en cada carpeta obedece la siguiente distribución.

- “Train” = 5000 imágenes por cada etiqueta.
- “Test” = 500 imágenes por cada etiqueta.

4.7 Entrenamiento y estadísticas

Para iniciar el entrenamiento es necesario estar colocado en la siguiente dirección:

```
/home/Jetson/train/
```

En terminal se iniciara el entrenamiento con el siguiente comando:

```
python train.py --model-dir=asphalt ~/datasets/asphalt --model=resnet18
--epochs 100 --batch_size=8 --input_shape=224 --channels=3
```

Esto deberá iniciar el entrenamiento, con lo cual se desplegaran las estadísticas en tiempo real del entrenamiento figura 4.6.

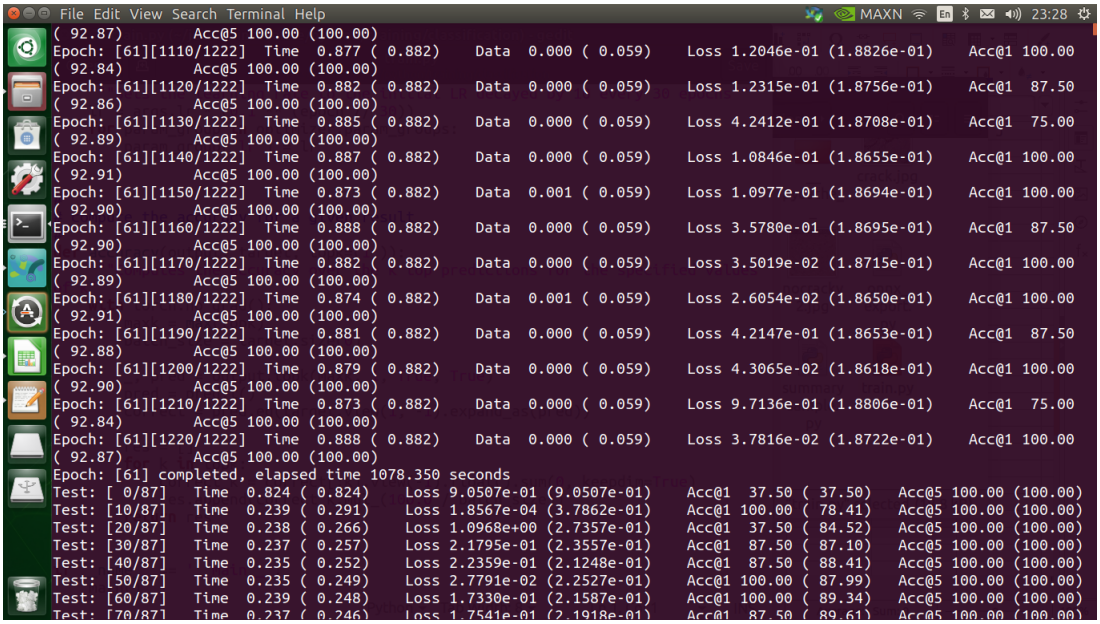


Figura 4.6: Entrenamiento en terminal

Cada iteración (Epoch) requiere 900 segundos de entrenamiento en el Jetson nano, cumplir con 100 iteraciones del modelo lleva aproximadamente 25 horas. Una vez concluido el entrenamiento podremos evaluar su desempeño a través de las estadísticas de entrenamiento.

4.7.1. Precisión

La precisión del modelo es el porcentaje en que nuestro modelo infirió de manera correcta la etiqueta de la imagen de entrada, en otras palabras el total de aciertos sobre el número de iteraciones, la ecuación 4.8 describe este cálculo.

$$P = \left(\frac{V_A - V_O}{V_A} \right) 100 \tag{4.8}$$

Donde V_A representa las observaciones aceptadas y V_O las observaciones obtenidas.

La precisión del modelo es calculada para dos conjuntos de datos, las inferencias realizadas para el entrenamiento y la validación, la figura 4.7 muestra la gráfica comparativa de la precisión obtenida. En ambas variables se muestra un incremento en la precisión del modelo, siendo un poco mas len-

to en la validación, donde las imágenes utilizadas no pertenecieron al entrenamiento, después de la iteración (epoch) 40 se logra una estabilización en ambas variables entre 85 % y 94 %, por lo que no existe evidencia suficiente para asumir que un entrenamiento de más iteraciones podría incrementar el rendimiento del modelo. La mayor precisión en la validación se logra en la iteración 70 con un porcentaje de 94.027, por lo cual el modelo generado en esta iteración será el modelo implementado para la detección inteligente de fallas en el pavimento.

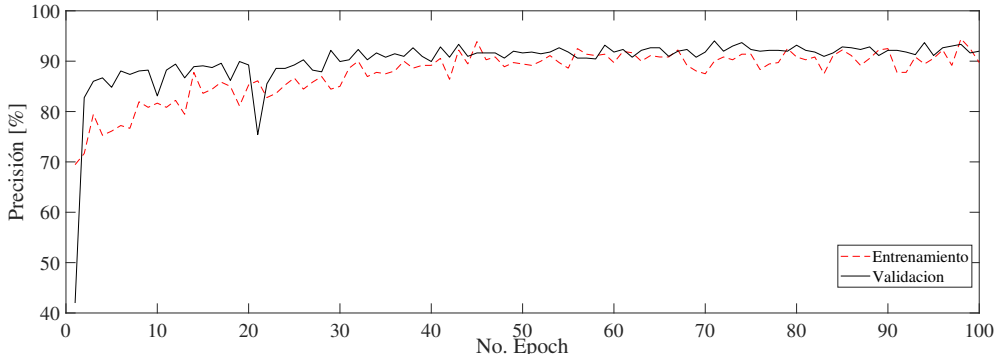


Figura 4.7: Comparativa precisión entrenamiento vs validación

4.7.2. Pérdida

En nuestro problema, al ser un problema de clasificación binaria, la pérdida puede ser calculado con función de la pérdida logarítmica, la cual evalúa cada predicción contra la etiqueta real del conjunto de entrada, una función logarítmica penaliza las diferencias grandes mientras que ofrece un resultado pequeño con diferencias mínimas, ecuación 4.9.

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij} \tag{4.9}$$

Donde N represente el número de muestras, M representa el número de etiquetas posibles, y_{ij} es un indicador binario sobre si la etiqueta j es o no correcta, finalmente p_{ij} es el modelo de probabilidad a la etiqueta j en la instancia i.

En nuestro entrenamiento obtenemos dos conjuntos de resultados para pérdida, uno en por la inferencia del conjunto de imágenes utilizado para el propio entrenamiento y otro con el conjunto de imágenes utilizado para la validación. La figura 4.8 muestra la gráfica para la pérdida de estas dos variantes a través de 100 iteraciones. Como se puede observar el resultado es similar, muestra un comportamiento logarítmico con oscilaciones a partir del punto de establecimiento, la pérdida en el entrenamiento es ligeramente menor debido a que las imágenes utilizadas en la validación difieren a las imágenes

utilizadas en el entrenamiento, el menor valor obtenido de pérdida es de 0.1655 en el entrenamiento, mientras que en la validación el menor valor es 0.2692.

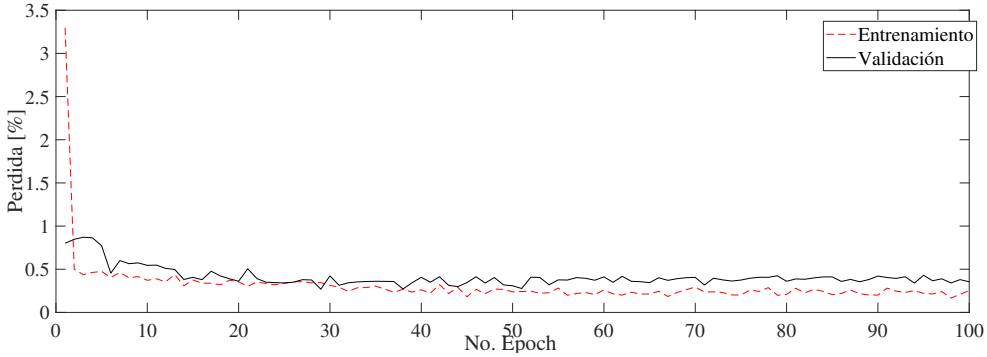


Figura 4.8: Comparativa pérdida entrenamiento vs validación

4.8 Conversión a ONNX

ONNX Open Neural Network Exchange por sus siglas en inglés, es un formato libre para representar modelos de aprendizaje profundo, esto permite la compatibilidad con hardware diverso. El modelo generado en la sección anterior posee una extensión .pth debido a que es un archivo generado por PyTorch, para poder utilizar el GPU disponible en el Jetson Nano, es necesario convertir este archivo a la extinción .onnx.

Lo anterior puede ser logrado utilizando un script de conversión apéndice A.2.1, para iniciar el script es necesario indicar los parámetros de la tabla 4.4

Tabla 4.4: Parámetros de conversión

Parámetro	Descripción
–input	Nombre del modelo entrenado a convertir
–output	Dirección donde se desea guardar el modelo
–model-dir	Dirección del modelo utilizado
–no-softmax	Agregar una capa nn.softmax

4.9 Inferencia en tiempo real

Para utilizar el modelo entrenado en tiempo real es necesario utilizar una cámara, como se describió en el capitulo 2 la cámara se comunicara por protocolo USB. Para conocer el Linux que la cámara esta correctamente configurada se utiliza el siguiente comando:

```
v4l2-ctl --list-devices
```

Correr este comando obtiene como respuesta un mensaje mostrado en la figura 4.9. Donde se indica la dirección del dispositivo de video disponible, en este caso /dev/video0.

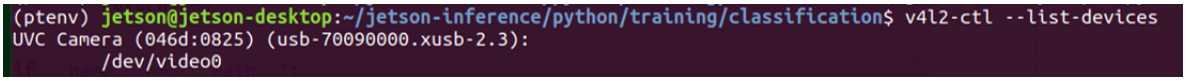


Figura 4.9: Lista de dispositivos disponibles

Nvidia proporciona un programa de python ”imagenet-camera.py” para correr modelos de inferencia utilizando una cámara como dispositivo de entrada, por lo cual podremos utilizar el modelo entrenado con esta aplicación. La tabla 4.5 muestra los parámetros para ejecutar este programa.

Tabla 4.5: Parámetros imagenet

Parámetro	Descripción
–Network	Nombre del modelo entrenado
–camera	Dirección de la cámara a utilizar
–width	Ancho deseado
–height	Altura deseada

El modelo entrenado es nombrado ”asphalt”, las etiquetas del modelo están contenidas en el archivo ”labels.txt”, la resolución de la cámara es de 1280 píxeles por 720 píxeles, mediante la utilización del siguiente comando podremos iniciar con la inferencia de nuestro modelo en tiempo real.

```
imagenet-camera.py --model=asphalt/aspahlt.onnx
--input_blob=input_0 --output_blob=output_0 --labels=asphaltv2/labels.txt
--camera=/dev/video0 --width=1280 --height=720
```

Se deberá mostrar una ventana con las imágenes de entrada así como la etiqueta correspondiente a la inferencia del modelo figura 4.10.

4.9.1. Costo computacional

Con el modelo de inferencia se logran realizar 90 inferencias por minuto, donde el cálculo de la tabla 4.1 indica que se realizan 1.8×10^9 operaciones de punto flotante por segundo por cada iteración, la cantidad de información generada excedería la capacidad de procesamiento de los procesadores de la computadora, por este motivo el modelo de inferencia es cargado directamente en el GPU de la computadora, el cual esta diseñado para el procesamiento numérico de gran escala.

En la figura 4.11 se refleja el costo computacional cuando la red neuronal convolucional de inferencia es ejecutada, es posible apreciar el porcentaje de utilización de cada uno de los núcleos del CPU, donde el porcentaje máximo es de 40 % en uno de los núcleos, el consumo de recursos en el GPU se mantiene variable con un máximo de 96 %.

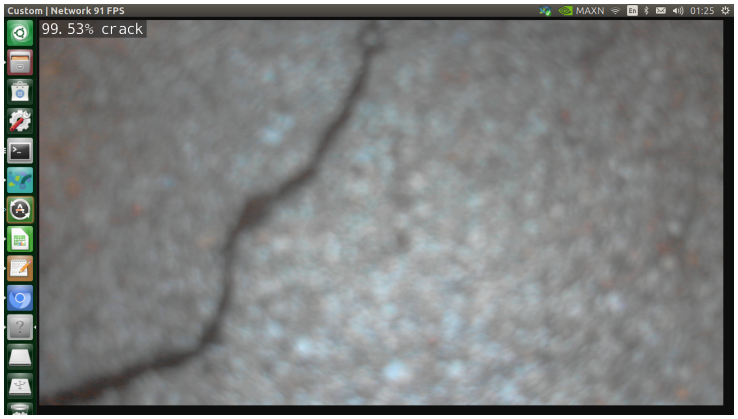


Figura 4.10: Inferencia en tiempo real

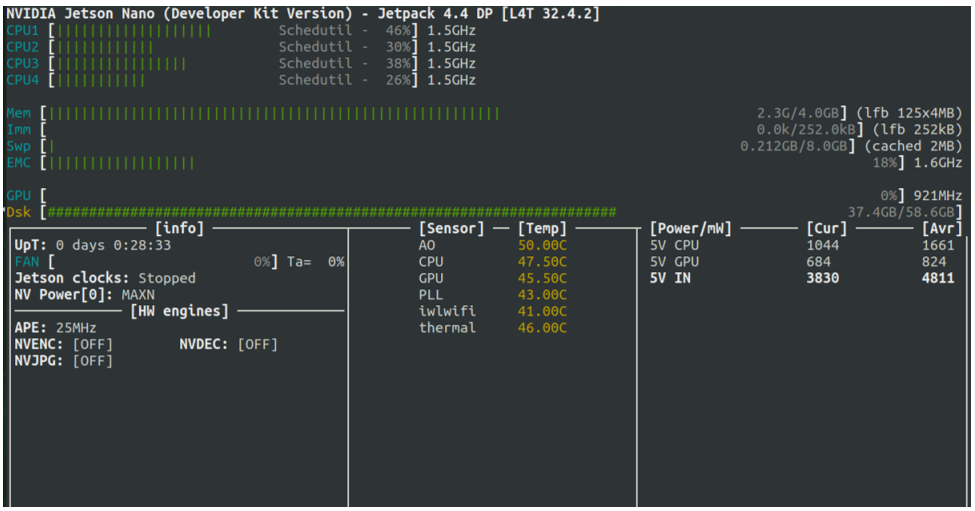


Figura 4.11: Consumo recursos-Inferencia

5. Resultados

5.1 Sistema de navegación basado en visión

El sistema de navegación desarrollado fue implementado en la plataforma robótica presentada en este proyecto, se realizó un experimento de navegación autónoma utilizando una ruta predeterminada, utilizando la estación en tierra mission Planner de la familia Autopilot, fue posible conocer la ubicación del robot obtenida por el sistema de navegación basado en visión en tiempo real. La figura 5.1 muestra el mensaje enviado por nuestro algoritmo a una frecuencia de 30 Hertz, la figura 5.2 muestra el experimento conducido, donde en la figura 5.2a apreciamos la ubicación real del robot y en la figura 5.2b visualizamos la representación del desplazamiento en Mission Planner.

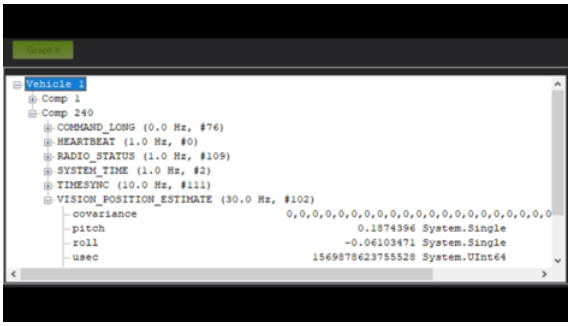


Figura 5.1: Recepción de odometría

(a) Real **(b) Representación**

Figura 5.2: Experimento de navegación

No es posible graficar la información de desplazamiento enviada a la computadora del vuelo por el algoritmo, sin embargo, el ecosistema Autopilot permite conocer y graficar diversos parámetros de nuestra plataforma, el sistema de navegación envía el desplazamiento en metros del robot, el ecosistema transforma ese desplazamiento en coordenadas geodésicas para determinar la ubicación del robot, dichas coordenadas son presentadas en las gráficas de las figuras 5.3 y 5.4 donde se gráfica la posición del robot en función del tiempo. Respecto a la orientación de la plataforma, al ser un robot terrestre solo modifica su orientación en el ángulo Yaw, la figura 5.5 muestra la aceleración angular en radianes sobre segundo de la plataforma durante el experimento, las oscilaciones representan un cambio de

orientación.

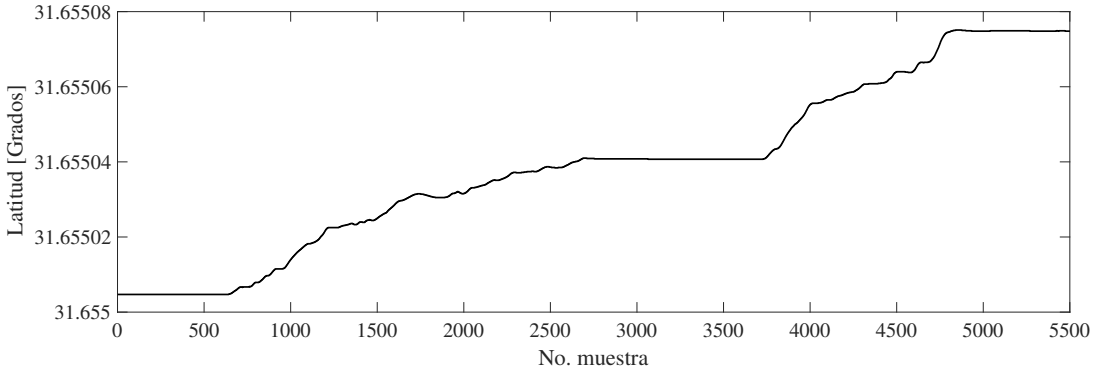


Figura 5.3: Cambio latitud

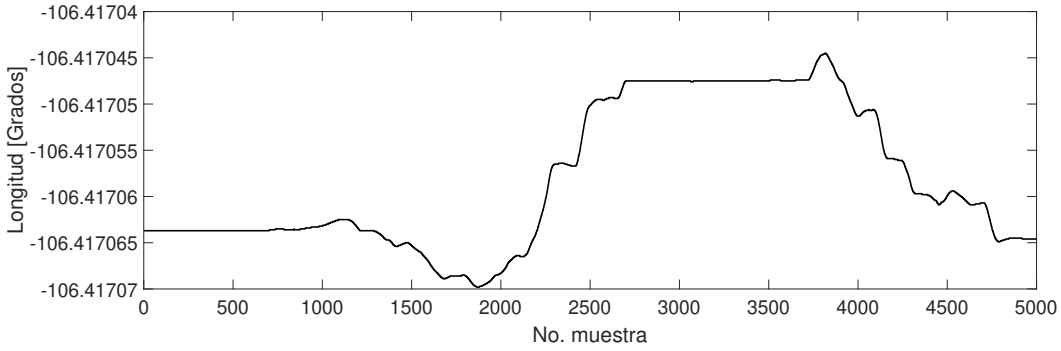


Figura 5.4: Cambio longitud

La comprobación del envió y recepción del mensaje creado por el algoritmo en la computadora de vuelo, nos permite validar el funcionamiento correcto del sistema, las gráficas mostradas en esta sección nos permiten validar la correcta conversión de datos de visión en desplazamientos, así como su correcta orientación, por lo cual se puede decir que el sistema de navegación basado en visión fue desarrollado e implementado correctamente.

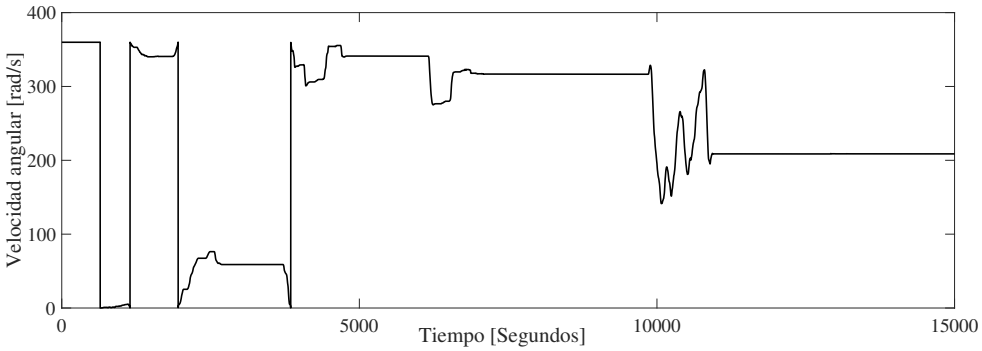


Figura 5.5: Ángulo Yaw

5.2 Resultados sistema de inspección de pavimento inteligente

Para evaluar nuestro modelo entrenado para la inspección inteligente de pavimento se utilizo un conjunto de imágenes alterna al proceso de entrenamiento, dicho conjunto posee imágenes para ambas etiquetas, los resultados obtenidos nos permiten calcular el resultado F1 (F1 score), el cual es la media entre la precisión (precisión) ecuación 5.1 y exhaustividad (recall) ecuación 5.2, la ecuación de dicho cálculo es definida por otros investigadores en la ecuación 5.3.

$$Pr = \frac{TP}{TP+FP} \tag{5.1}$$

$$Re = \frac{TP}{TP+FN} \tag{5.2}$$

$$F1 = \frac{2PrRe}{Pr+Re} \tag{5.3}$$

Donde Pr es la precisión del modelo, TP son verdaderos positivos, FP falsos positivos, FN falsos negativos, Re es la exhaustividad y $F1$ es la media armónica entre precisión y exhaustividad. Con las inferencias realizadas se construye la matriz de confusión 5.1. De 100 inferencias para imágenes con grietas, se obtuvieron 92 resultados positivos mientras que 8 mostraron un falso positivo, para 102 inferencias sin grietas 98 casos fueron negativos y 4 falsos negativos. En base de estos resultados obtenemos el resultado F1 para la inferencia de nuestro modelo para ambas etiquetas tabla 5.2. La precisión para inferir que la imagen de entrada no contiene grietas es 0.04 mayor que la inferencia de que posee grietas.

Tabla 5.1: Matriz de confusión

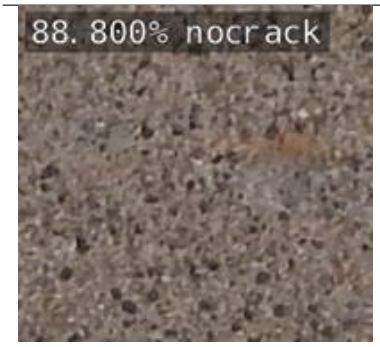
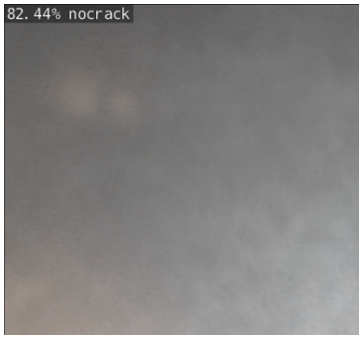
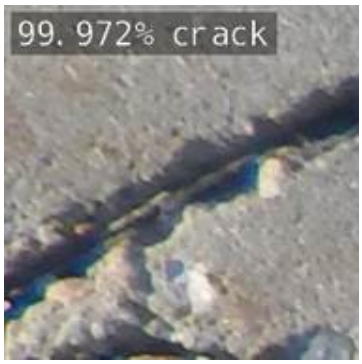
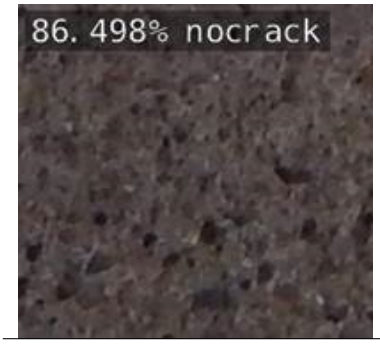
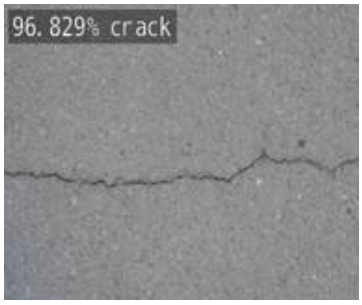
		Predicción	
		Grieta	Sin grieta
Realidad	Grieta	92	98
	Sin grieta	8	4

Tabla 5.2: Resultado F1

Parámetro	**Grieta**	**Sin grieta**
Precisión	0.92	0.96
Recall	0.958	0.924
F1	0.938	0.942

La tabla 5.3 muestra ejemplos de las imágenes de entrada y las inferencias hechas por la red neuronal entrenada.

Tabla 5.3: Resultados inferencias

Inferencia	Resultado	Inferencia	Resultado
	0.88800 Sin Grieta		0.99943 Grieta
	0.8244 Sin Grieta		0.99972 Grieta
	0.86498 Sin Grieta		0.96829 Grieta

5.2.1. Consumo computacional del sistema

En la etapa de experimentación fue posible ejecutar ambos subsistemas de manera paralela, esto gracias a que la principal carga del algoritmo de navegación esta ubicada en el procesador mientras que el sistema de detección inteligente ejecuta su modelo entrenado en el GPU. La figura 5.8 muestra la distribución de carga computacional, los 4 núcleos de la computadora llegan hasta el 80 %, mientras que el GPU mantiene su comportamiento de oscilación al igual que cuando es ejecutado de manera individual. La figura 5.6 muestra un comportamiento estable en los cuatro núcleos del procesador, mientras que la figura 5.7 muestra la oscilación mencionada, sin embargo se muestra una tendencia a mantenerse en el rango del 90 % de utilización, lo que no permitiría ejecutar mas tareas que requieran del GPU.

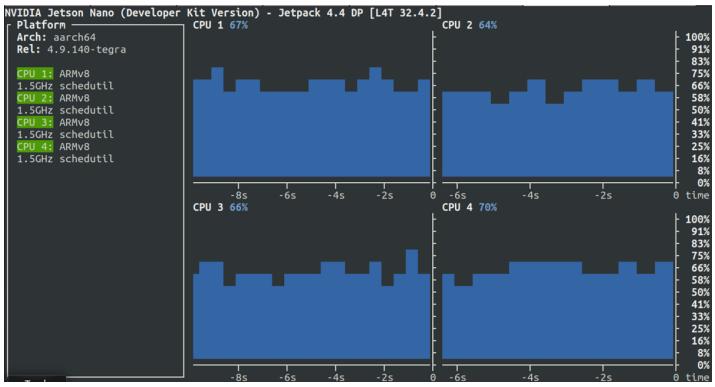


Figura 5.6: Carga CPU

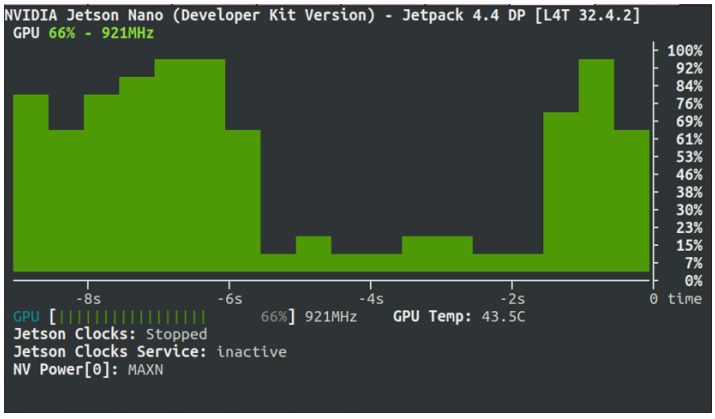


Figura 5.7: Carga GPU

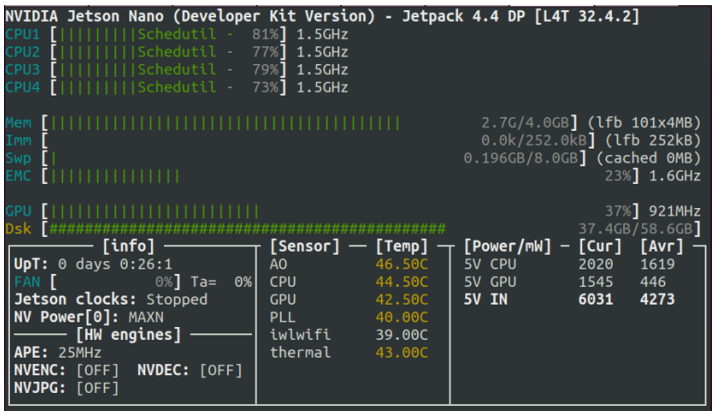


Figura 5.8: Consumo computacional

6. Conclusiones y líneas futuras

6.1 Conclusiones

Referente a los objetivos trazados en este proyecto, se ha conseguido implementar un sistema de navegación basado en visión en un vehículo a escala, quien provee la capacidad de calcular el desplazamiento del vehículo en un marco de referencia y por ende ubicar en el espacio al robot sin previo conocimiento del entorno, por lo cual la navegación autónoma que prescinde del uso de GPS fue lograda.

Respecto al desarrollo e implementación de un sistema de detección inteligente de fallas, se pudieron implementar los principios de inteligencia artificial, específicamente redes neuronales convolucionales las cuales permitieron el desarrollo de un sistema de clasificación binario para detección de fallas en el pavimento, a pesar del costo computacional que representa correr un algoritmo de inferencia en tiempo real esto fue logrado en el vehículo a escala.

Ambos sistemas fueron validados obteniendo resultados satisfactorios, aun así existen mejoras que se pueden realiza a ambos sistemas.

En el sistema de navegación basado en visión, la velocidad de muestreo del sistema esta limitada a 30 hertz, debido a esto en movimientos a alta velocidad podría no ser suficiente, una futura implementación con una computadora de vuelo de mayores prestaciones permitirá utilizar los 200 hertz en los que puede ser calculado el desplazamiento de la cámara T265. Así mismo la integración de otros sensores puede reducir la cantidad de pasos manuales antes de iniciar una misión autónoma, tales como la integración de un magnetómetro.

En el sistema de detección inteligente se utilizo una red compuesta por 18 capaz, dicho enfoque estuvo limitado por las prestaciones del hardware disponible, una computadora con mejores prestaciones permitirá realizar entrenamientos de más capaz en menor tiempo, por lo que no se descarta un mejor desempeño del modelo con un entrenamiento mas robusto. Así mismo, el sistema de clasificación implementado solo brinda información si una falla esta presente o no, un sistema de detección de objetos permitirá conocer además de la presencia de una falla, su ubicación exacta, sin embargo, un sistema de clasificación es computacional mente liviano relativamente, un sistema de detección supone un reto mayor tanto para su desarrollo como en la implementación en una computadora como lo es Jetson nano.

6.2 Líneas futuras

El potencial de la aplicación de inteligencia artificial en la robótica es amplio, conocer mas información del entorno permitirá a un robot móvil modificar su comportamiento para un mejor rendimiento, sistemas de detección de objetos mas robusto brindará mas certidumbre del entorno.

Otro linea es la integración de una gama de sensores que complementen al sistema de navegación basado en visión para robustecer las capacidades del mismo, sensores de vanguardia como del tipo Lidar permitirán desarrollar sistemas con mejores prestaciones.

Referencias

Aamer, N., y Ramachandran, S. (2015). Neural Networks Based Addaptive Approach for Path Planning and Obstacle Avoindance for Autonomous Mobile Robots (AMR). *International Journal of Reserach in Computer Applications and Robotics*, *3*(12), 66–79.

Ab Ghani, N., Abidin, S. Z. Z., y Zamani, M. (2011). Generating Transition Rules of Cellular Automata for Urban Growth Prediction. *International Journal of Geology*, *5*(2), 41–47.

Aulinas, J., Petillot, Y., Salvi, J., y Lladó, X. (2008). The SLAM problem: A survey. *Frontiers in Artificial Intelligence and Applications*, *184*(1), 363–371. doi: 10.3233/978-1-58603-925-7-363

autopilot, P. (2019). *External position estimation.* Descargado de `https://dev.px4.io/v1.9.0/en/ros/` $external_position_estimation.html using-vision-or-motion-capture-systems-for-position-estimation$

Bambino, I. (2008). Una Introducción a los Robots Móviles. *Una Introducción a los Robots Móviles Il*, 1–86.

Bergerman, M., Maeta, S. M., Zhang, J., Freitas, G. M., Hamner, B., Singh, S., y Kantor, G. (2015). Robot farmers: Autonomous orchard vehicles help tree fruit production. *IEEE Robotics and Automation Magazine*, *22*(1), 54–63. doi: 10.1109/MRA.2014.2369292

Brahmanage, G., y Leung, H. (2017). A kinect-based SLAM in an unknown environment using geometric features. *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, *2017-Novem*(Mfi), 570–575. doi: 10.1109/MFI.2017.8170382

Burgner-Kahrs, J., Rucker, D. C., y Choset, H. (2015). *Continuum Robots for Medical Applications: A Survey* (Vol. 31) (n.º 6). IEEE. doi: 10.1109/TRO.2015.2489500

Capua, F. R., Nacional, U., y Moreyra, M. L. (2018). Comparative analysis of Visual-SLAM algorithms applied to fruit environments. *Argentine Association of Automatic Control*, 1–6.

Chen, Z., Samarabandu, J., y Rodrigo, R. (2007). Recent advances in simultaneous localization and map-building using computer vision. *Advanced Robotics*, *21*(3-4), 233–265. doi: 10.1163/156855307780132081

Corke, P. (2017). *Robotics, vision and control* (Vol. 118). doi: 10.1007/978-3-319-54413-7

DeSouza, G. N., y Kak, A. C. (2002). Vision for mobile robot navigation: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *24*(2), 237–267. doi: 10.1109/34.982903

Dorafshan, S., Thomas, R. J., y Maguire, M. (2018). SDNET2018: An annotated image dataset for non-contact concrete crack detection using deep convolutional neural networks. *Data in Brief*, *21*, 1664–1668. Descargado de `https://doi.org/10.1016/j.dib.2018.11.015` doi: 10.1016/j.dib.2018.11.015

Dung, C. V., y Anh, L. D. (2019). Autonomous concrete crack detection using deep fully convolutional neural network. *Automation in Construction*, *99*(October 2018), 52–58. Descargado de `https://doi.org/10.1016/j.autcon.2018.11.028` doi: 10.1016/j.autcon.2018.11.028

Emily, L. P., Huat, S., Poh, L., Toh, E., Causo, A., Tzuo, P. W., … Yeo, S. H. (2016). A Review on the Use of Robots in Education and Young Children. , *17*(4).

Endres, F., Hess, J., Engelhard, N., Sturm, J., Cremers, D., y Burgard, W. (2012). An evaluation of the RGB-D SLAM system. *Proceedings - IEEE International Conference on Robotics and Automation*(October 2014), 1691–1696. doi: 10.1109/ICRA.2012.6225199

Filipenko, M., y Afanasyev, I. (2018). Comparison of Various SLAM Systems for Mobile Robot in an Indoor Environment. *9th International Conference on Intelligent Systems 2018: Theory, Research and Innovation in Applications, IS 2018 - Proceedings*(September), 400–407. doi: 10.1109/IS.2018.8710464

Findler, N. V. (2007). ARTIFICIAL INTELLIGENCE. *The Journal of the Emeritus College at ASU*, *1*.

Gage, D. W. (1995). Special Issue on Unmanned Ground Vehicles FIGURES OMITTED Unmanned Systems Magazine. , *13*(3). Descargado de `http://www.dtic.mil/dtic/tr/fulltext/u2/a422845.pdf`

Gamboa, P. (2019). *Hoyos y baches abundan en las calles de Juárez.* Cd. Juarez. Descargado de `https://www.elheraldodejuarez.com.mx/local/hoyos-y-baches-abundan-en-las-calles-de-juarez-4270911.html`

Givigi, S. N., Freitas, A. D. M., y Beaulieu, A. (2018). Autonomous Robot System for Inspection of Defects in Civil Infrastructures. , *12*(2), 1414–1422. doi: 10.1109/JSYST.2016.2611244

Google. (2020). *Google maps.* Descargado de `https://www.google.com.mx/maps/@31.7430575,-106.43092,165m/data=!3m1!1e3`

Gopalakrishnan, K., Khaitan, S. K., Choudhary, A., y Agrawal, A. (2017). Deep Convolutional Neural Networks with transfer learning for computer vision-based data-driven pavement distress detection. *Construction and Building Materials*, *157*, 322–330. Descargado de `https://doi.org/10.1016/j.conbuildmat.2017.09.110` doi: 10.1016/j.conbuildmat.2017.09.110

Hayet, T., Hatem, T., y Jilani, K. (2016). Navigation Control of a Mobile Robot under Time Constraint using Genetic Algorithms, CSP Techniques, and Fuzzy Logic. *Nature-Inspired Computing*, 932–954. doi: 10.4018/978-1-5225-0788-8.ch035

Henry, P., Krainin, M., Herbst, E., Ren, X., y Fox, D. (2012). RGB-D mapping: Using Kinect-style depth cameras for dense 3D modeling of indoor environments. *International Journal of Robotics Research*, *31*(5), 647–663. doi: 10.1177/0278364911434148

Holybro. (2019). *Pixhawk.* Descargado de `https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html`

Ibramigov, I. Z., y Afanasyev, I. M. A. (2017). Comparison of ROS-based Visual SLAM methods in homogeneous indoor environment. *Workshop on Positioning, Navigation and Communications*.

Intel. (2019a). *Intel® RealSense™ Depth Camera D435.* Descargado de `https://www.intelrealsense.com/ depth-camera-d435/`

Intel. (2019b). *Intel® RealSense™ Tracking Camera T265.* Descargado de `https://www.intelrealsense.com/tracking-camera-t265/?utm_source=intelcom_website&utm_ medium=button&utm_campaign=launch_T265&`

utm _content=T265_learn-more_button

Juarez, C. (2019). *Daños por baches, reportan pocos y mal.* Descargado de https://netnoticias.mx/juarez/ danos-por-baches-reportan-pocos-y-mal/

Kaiming, Zhang, X., Ren, S., y Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, *2016-December*, 770–778. doi: 10.1109/CVPR.2016.90

Kerl, C., Sturm, J., y Daniel, C. (2013). Robust Odometry Estimation for RGB-D Cameras. , *3*, 427–436. doi: 10.7198/S2237-0722201300050035

Kim, J., y Sukkarieh, S. (2004). Autonomous airborne navigation in unknown terrain environments. *IEEE Transactions on Aerospace and Electronic Systems*, *40*(3), 1031–1045. doi: 10.1109/TAES.2004.1337472

Kimon P, V., y Vachtsevanos, G. J. (2015). *Handbook of Unmanned Aerial Vehicles*.

Kragic, D., y Vincze, M. (2009). Vision for Robotics. *Foundations and Trends in Robotics*, *1*(1), 1–78. doi: 10.1561/2300000001

Kumra, S., y Kanan, C. (2017). Robotic grasp detection using deep convolutional neural networks. *IEEE International Conference on Intelligent Robots and Systems*, *2017-Septe*, 769–776. doi: 10.1109/IROS.2017.8202237

Lecun, Y., Bengio, Y., y Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. doi: 10.1038/nature14539

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., y Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *International Journal of Robotics Research*, *37*(4-5), 421–436. doi: 10.1177/0278364917710318

Li, C., Wei, H., y Lan, T. (2017). Research and implementation of 3D SLAM algorithm based on Kinect depth sensor. *Proceedings - 2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics, CISP-BMEI 2016*, 1070–1074. doi: 10.1109/CISP-BMEI.2016.7852872

Liu, Y., Zhu, M., y Zhang, H. (2017). Processed RGB-D Slam Using Open-Source Software. *Proceedings - 2017 IEEE International Conference on Computational Science and Engineering and IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, CSE and EUC 2017*, *1*, 623–626. doi: 10.1109/CSE-EUC.2017.115

Lopez Torres, P. (2016). Análisis De Algoritmos Para Localización Y Mapeado Simultáneo De Objetos. , 51. Descargado de https://idus.us.es/xmlui/handle/11441/54393

Mathiassen, K. (2016). An Ultrasound Robotic System Using the Commercial Robot UR5. , *3*(February), 1–16. doi: 10.3389/frobt.2016.00001

Matsas, E., Vosniakos, G.-c., y Batras, D. (2018). Robotics and Computer – Integrated Manufacturing Prototyping proactive and adaptive techniques for human-robot collaboration in manufacturing using virtual reality. , *50*(October 2016), 168–180. doi: 10.1016/j.rcim.2017.09.005

Ming-hui, Z., Sai, M. A., y Xing-ru, W. (2017). Design of Moving System for Mining Searching Robot. , 3–5. doi: 10.1109/IHMSC.2017.15

Motsch, J., Benammar, S., y Bergeon, Y. (2017). Interior mapping of a building: A real-life experiment with Microsoft Kinect for Windows v1 and RGBD-SLAM. *ICMT 2017 - 6th International Conference on Military Technologies*, 728–732. doi: 10.1109/MILTECHS.2017.7988852

Nvidia. (2019). *Jetson Nano.* Descargado de `https://www.nvidia.com/es-la/autonomous-machines/ embedded-systems/jetson-nano/`

Olmos, J. (2020). *Tapan baches, se abren otros.* Cd. Juarez. Descargado de `https://diario.mx/juarez/tapan-baches-se-abren-otros-20200109-1613306.html`

Onal, H. (2008). Simple relations between elements of the three-dimensional orthogonal matrix of the basic representation and euler rpy' angles. *IBSU Scientific Journal*, *2*(1), 187–192. Descargado de `https://journal.ibsu.edu.ge/index.php/ibsusj/article/view/40`

Ortiz, S., Yu, W., y Zamora, E. (2019). Sliding Mode SLAM for Robust Simultaneous Localization and Mapping. *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, *1*, 5674–5679. doi: 10.1109/iecon.2018.8591121

Powell, V. (2018). *Image Kernels.* Descargado de `https://setosa.io/ev/image-kernels/`

Priyandoko, G., Ming, T. Y., y Achmad, M. S. (2017). Mapping of unknown industrial plant using ROS-based navigation mobile robot. *IOP Conference Series: Materials Science and Engineering*, *257*(1). doi: 10.1088/1757-899X/257/1/012088

Ren, X., Bo, L., y Fox, D. (2012). RGB-(D) scene labeling: Features and algorithms. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*(D), 2759–2766. doi: 10.1109/CVPR.2012.6247999

Rosebrock, A. (2017). Deep Learning for Computer Vision with Python. *PyImageSearch*, *53*, 160. doi: 10.1017/CBO9781107415324.004

Rubiales, A. (2019). *Regresión lineal.* Descargado de

Samuel, A. L. (1959). Some Studies in Machine Learning. *IBM Journal of Research and Development*, *3*(3), 210–229.

Shapiro, L., y Stockman, G. (2001). *Computer vision*. Descargado de `http://repositorio.unan.edu.ni/2986/1/5624.pdf`

Shehata, H. M., Mohamed, Y. S., Abdellatif, M., y Awad, T. H. (2018). Depth estimation of steel cracks using laser and image processing techniques. *Alexandria Engineering Journal*, *57*(4), 2713–2718. Descargado de `https://doi.org/10.1016/j.aej.2017.10.006` doi: 10.1016/j.aej.2017.10.006

Shnayderman, A., Gusev, A., y Eskicioglu, A. M. (2006). An SVD-based grayscale image quality measure for local and global assessment. *IEEE Transactions on Image Processing*, *15*(2), 422–429. doi: 10.1109/TIP.2005.860605

Shvets, A. A., Rakhlin, A., Kalinin, A. A., y Iglovikov, V. I. (2019). Automatic Instrument Segmentation in Robot-Assisted Surgery using Deep Learning. *Proceedings - 17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018*, 624–628. doi: 10.1109/ICMLA.2018.00100

Siegwart, R., y Nourbakhsh, I. (2004). *A introduction to autonomous mobile robots*. IEEE. doi: 10.1038/nn0804-796

Sivakorn, S., Polakis, I., y Keromytis, A. D. (2016). I am Robot: (Deep) learning to break semantic image CAPTCHAs. *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, 388–403. doi: 10.1109/EuroSP.2016.37

Traxxas. (2020). *Traxxas Rustler*. Descargado de `https://traxxas.com/products/models/electric/37054-1rustler`

Vazquez Cespedes, H. (2011). Robotica: Posiciones y marcos de referencia. *Universidad de Costa Rica, Ingenieria*.

Víctor, R., Barrientos, S., José, R., García, S., Ramón, S., y Ortigoza, APACrefauthors (2007). Robots Móviles: Evolución y Estado del Arte. Descargado de `https://www.polibits.gelbukh.com/2007_35/ Robots Moviles_ Evolucion y Estado del Arte.pdf`

Wang, J., y Wilson, W. (1992, may). 3d relative position and orientation estimation using kalman filter for robot control. , 2638,2639,2640,2641,2642,2643,2644,2645. Descargado de `https://doi.ieeecomputersociety.org/10.1109/ROBOT.1992.220044` doi: 10.1109/ROBOT.1992.220044

Wu, Y., y Ding, Z. (2018). Research on Laser Navigation Mapping and Path Planning of Tracked Mobile Robot Based on Hector SLAM. *2018 International Conference on Intelligent Informatics and Biomedical Sciences, ICIIBMS 2018*, *3*, 59–65. doi: 10.1109/ICIIBMS.2018.8549954

Zaffar, M., Ehsan, S., Stolkin, R., y Maier, K. M. D. (2018). Sensors, SLAM and Long-term Autonomy: A Review. *2018 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2018*, 285–290. doi: 10.1109/AHS.2018.8541483

Zanagnolo, V., Garbi, A., Achilarre, M. T., y Minig, L. (2017). *Robot-assisted Surgery in Gynecologic Cancers* (Vol. 24) (n.º 3). Elsevier Ltd. Descargado de `http://dx.doi.org/10.1016/j.jmig.2017.01.006` doi: 10.1016/j.jmig.2017.01.006

Zhang, L., Yang, F., Daniel Zhang, Y., y Zhu, Y. J. (2016). Road crack detection using deep convolutional neural network. *Proceedings - International Conference on Image Processing, ICIP*, *2016-August*, 3708–3712. doi: 10.1109/ICIP.2016.7533052

A. Apéndice

A.1 Imágenes

A.1.1. Entradas y salidas Jetson Nano

Jetson Nano J41 Header					
Sysfs GPIO	Name	Pin	Pin	Name	Sysfs GPIO
	3.3 VDC *Power*	1	2	**5.0 VDC** *Power*	
	I2C_2_SDA *I2C Bus 1*	3	4	**5.0 VDC** *Power*	
	I2C_2_SCL *I2C Bus 1*	5	6	**GND**	
gpio216	**AUDIO_MCLK**	7	8	**UART_2_TX** */dev/ttyTHS1*	
	GND	9	10	**UART_2_RX** */dev/ttyTHS1*	
gpio50	**UART_2_RTS**	11	12	**I2S_4_SCLK**	gpio79
gpio14	**SPI_2_SCK**	13	14	**GND**	
gpio194	**LCD_TE**	15	16	**SPI_2_CS1**	gpio232
	3.3 VDC *Power*	17	18	**SPI_2_CS0**	gpio15
gpio16	**SPI_1_MOSI**	19	20	**GND**	
gpio17	**SPI_1_MISO**	21	22	**SPI_2_MISO**	gpio13
gpio18	**SPI_1_SCK**	23	24	**SPI_1_CS0**	gpio19
	GND	25	26	**SPI_1_CS1**	gpio20
	I2C_1_SDA *I2C Bus 0*	27	28	**I2C_1_SCL** *I2C Bus 0*	
gpio149	**CAM_AF_EN**	29	30	**GND**	
gpio200	**GPIO_PZ0**	31	32	**LCD_BL_PWM**	gpio168
gpio38	**GPIO_PE6**	33	34	**GND**	
gpio76	**I2S_4_LRCK**	35	36	**UART_2_CTS**	gpio51
gpio12	**SPI_2_MOSI**	37	38	**I2S_4_SDIN**	gpio77
	GND	39	40	**I2S_4_SDOUT**	gpio78

Figura A.1: Entradas y salidas Jetson Nano

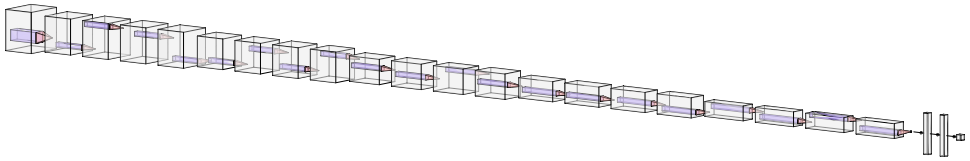


Figura A.2: Representación visual del modelo

A.1.2. Modelo Resnet18

A.2 Código

A.2.1. Código conversión ONNX

```
import os
import argparse

import torch
import torchvision.models as models

from reshape import reshape_model

model_names = sorted(name for name in models.__dict__
    if name.islower() and not name.startswith("__")
    and callable(models.__dict__[name]))

# parse command line
parser = argparse.ArgumentParser()
parser.add_argument('--input', type=str, default='model_best.pth.tar',
help="path to input PyTorch model (default: model_best.pth.tar)")
parser.add_argument('--output', type=str, default='', help="desired path
of converted ONNX model (default: <ARCH>.onnx)")
parser.add_argument('--model-dir', type=str, default='', help="directory
to look for the input PyTorch model in, and export the converted ONNX
model to (if --output doesn't specify a directory)")
parser.add_argument('--no-softmax', type=bool, default=False,
help="disable adding nn.Softmax layer to model (default is to add
Softmax)")

opt = parser.parse_args()
print(opt)
```

```
# format input model path
if opt.model_dir:
opt.model_dir = os.path.expanduser(opt.model_dir)
opt.input = os.path.join(opt.model_dir, opt.input)

# set the device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print('running on device ' + str(device))

# load the model checkpoint
print('loading checkpoint:  ' + opt.input)
checkpoint = torch.load(opt.input)
arch = checkpoint['arch']

# create the model architecture
print('using model:  ' + arch)
model = models.__dict__[arch](pretrained=True)

# reshape the model's output
model = reshape_model(model, arch, checkpoint['num_classes'])

# load the model weights
model.load_state_dict(checkpoint['state_dict'])

# add softmax layer
if not opt.no_softmax:
print('adding nn.Softmax layer to model...')
model = torch.nn.Sequential(model, torch.nn.Softmax(1))

model.to(device)
model.eval()

print(model)

# create example image data
resolution = checkpoint['resolution']
input = torch.ones((1, 3, resolution, resolution)).cuda()
print('input size:  {:d}x{:d}'.format(resolution, resolution))
```

```
# format output model path
if not opt.output:
opt.output = arch + '.onnx'

if opt.model_dir and opt.output.find('/') == -1 and
opt.output.find('\\') == -1:
opt.output = os.path.join(opt.model_dir, opt.output)

# export the model
input_names = [ "input_0" ]
output_names = [ "output_0" ]

print('exporting model to ONNX...')
torch.onnx.export(model, input, opt.output, verbose=True,
input_names=input_names, output_names=output_names)
print('model exported to:  {:s}'.format(opt.output))
```

A.2.2. Código rs t265.launch

```
<!--
Important Notice: For wheeled robots, odometer input is a requirement
for robust
and accurate tracking. The relevant APIs will be added to librealsense
and
ROS/realsense in upcoming releases. Currently, the API is available in
the
https://github.com/IntelRealSense/
librealsense/blob/master/third-party/
libtm/libtm/include/TrackingDevice.h#L508-L515.
-->
<launch>
  <arg name="serial_no"           default=""/>
  <arg name="usb_port_id"         default=""/>
  <arg name="device_type"         default="t265"/>
  <arg name="json_file_path"      default=""/>
  <arg name="camera"              default="camera"/>
  <arg name="tf_prefix"           default="$(arg camera)"/>

  <arg name="fisheye_width"       default="848"/>
```

```
  <arg name="fisheye_height"      default="800"/>
  <arg name="enable_fisheye1"     default="false"/>
  <arg name="enable_fisheye2"     default="false"/>

  <arg name="fisheye_fps"         default="30"/>

  <arg name="gyro_fps"            default="200"/>
  <arg name="accel_fps"           default="62"/>
  <arg name="enable_gyro"         default="true"/>
  <arg name="enable_accel"        default="true"/>

  <arg name="enable_sync"           default="false"/>

  <arg name="linear_accel_cov"      default="0.01"/>
  <arg name="initial_reset"         default="false"/>
  <arg name="unite_imu_method"      default=""/>

  <arg name="publish_odom_tf"     default="true"/>

  <group ns="$(arg camera)">
    <include file="$(find realsense2_camera)/launch/includes/
    nodelet.launch.xml">
      <arg name="tf_prefix"               value="$(arg tf_prefix)"/>
      <arg name="serial_no"               value="$(arg serial_no)"/>
      <arg name="usb_port_id"             value="$(arg usb_port_id)"/>
      <arg name="device_type"             value="$(arg device_type)"/>
      <arg name="json_file_path"          value="$(arg
json_file_path)"/>

      <arg name="enable_sync"             value="$(arg enable_sync)"/>

      <arg name="fisheye_width"           value="$(arg
fisheye_width)"/>
      <arg name="fisheye_height"          value="$(arg
fisheye_height)"/>
      <arg name="enable_fisheye1"         value="$(arg
enable_fisheye1)"/>
      <arg name="enable_fisheye2"         value="$(arg
enable_fisheye2)"/>
```

```
      <arg name="fisheye_fps"              value="$(arg fisheye_fps)"/>
      <arg name="gyro_fps"                 value="$(arg gyro_fps)"/>
      <arg name="accel_fps"                value="$(arg accel_fps)"/>
      <arg name="enable_gyro"              value="$(arg enable_gyro)"/>
      <arg name="enable_accel"             value="$(arg enable_accel)"/>

      <arg name="linear_accel_cov"         value="$(arg
linear_accel_cov)"/>
      <arg name="initial_reset"            value="$(arg
initial_reset)"/>
      <arg name="unite_imu_method"         value="$(arg
unite_imu_method)"/>

      <arg name="publish_odom_tf"          value="$(arg
publish_odom_tf)"/>
    </include>
  </group>
</launch>
```

A.2.3. Código Resnet18

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3,
3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
```

```
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
```

```
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
    (fc1): Linear(in_features=512, out_features=100, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=100, out_features=2, bias=True)
    (output): LogSoftmax()
  )
)
```

A.2.4. Código sistema de navegación

```
import sys
```

```
sys.path.append("/usr/local/lib/")

# Set MAVLink protocol to 2.
import os
os.environ["MAVLINK20"] = "1"

#  Libraries
import pyrealsense2 as rs
import numpy as np
import transformations as tf
import math as m
import time
import argparse
import threading
from time import sleep
from apscheduler.schedulers.background import BackgroundScheduler

from dronekit import connect, VehicleMode
from pymavlink import mavutil

########################################
# Parameters
########################################

# Default configurations for connection to the FCU
connection_string_default = '/dev/ttyUSB0'
connection_baudrate_default = 921600
connection_timeout_sec_default = 5


enable_msg_vision_position_estimate = True
vision_position_estimate_msg_hz_default = 30

# https://mavlink.io/en/messages/ardupilotmega.html#VISION_POSITION_DELTA
enable_msg_vision_position_delta = False
vision_position_delta_msg_hz_default = 30

# https://mavlink.io/en/messages/common.html#VISION_SPEED_ESTIMATE
enable_msg_vision_speed_estimate = True
```

```
vision_speed_estimate_msg_hz_default = 30

# https://mavlink.io/en/messages/common.html#STATUSTEXT
enable_update_tracking_confidence_to_gcs = True
update_tracking_confidence_to_gcs_hz_default = 1

# Default global position for EKF home/ origin
enable_auto_set_ekf_home = False
home_lat = 31743254    # Cd. Juarez Lat
home_lon = -106430562     # Cd. Juarez Lon
home_alt = 1120       # Cd. Juarez Alt

# TODO: Taken care of by ArduPilot, so can be removed (once the handling
on AP side is confirmed stable)
# In NED frame, offset from the IMU or the center of gravity to the
camera's origin point
body_offset_enabled = 0
comp_x = 0.225  # In meters (m)
comp_y = 0  # In meters (m)
comp_z = 0  # In meters (m)

# Global scale factor, position x y z will be scaled up/down by this
factor
scale_factor = 1.0



# pose data confidence: 0x0 - Failed / 0x1 - Low / 0x2 - Medium / 0x3 -
High
pose_data_confidence_level = ('FAILED', 'Low', 'Medium', 'High')

# lock for thread synchronization
lock = threading.Lock()

########################################
#  variables
########################################

# FCU variables
```

```
vehicle = None
is_vehicle_connected = False

# Camera variables
pipe = None
pose_sensor = None
linear_accel_cov = 0.01
angular_vel_cov  = 0.01

# variables
data = None
prev_data = None
H_aeroRef_aeroBody = None
V_aeroRef_aeroBody = None
heading_north_yaw = None
current_confidence_level = None
current_time_us = 0

# Increment everytime pose_jumping or relocalization happens
# See here: https://github.com/IntelRealSense/librealsense/blob/master/doc/t265.md#are-
# For AP, a non-zero "reset_counter" would mean that we could be sure
that the user's setup was using mavlink2
reset_counter = 1



parser = argparse.ArgumentParser(description='Reboots vehicle')
parser.add_argument('--connect',
                    help="Vehicle connection target string. If not
specified, a default string will be used.")
parser.add_argument('--baudrate', type=float,
                    help="Vehicle connection baudrate. If not specified,
a default value will be used.")
parser.add_argument('--vision_position_estimate_msg_hz', type=float,
                    help="Update frequency for VISION_POSITION_ESTIMATE
message. If not specified, a default value will be used.")
parser.add_argument('--vision_position_delta_msg_hz', type=float,
                    help="Update frequency for VISION_POSITION_DELTA
message. If not specified, a default value will be used.")
```

```
parser.add_argument('--vision_speed_estimate_msg_hz', type=float,
                    help="Update frequency for VISION_SPEED_DELTA
message. If not specified, a default value will be used.")
parser.add_argument('--scale_calib_enable', default=False,
action='store_true',
                    help="Scale calibration. Only run while NOT in
flight")
parser.add_argument('--debug_enable',type=int,
                    help="Enable debug messages on terminal")

args = parser.parse_args()

connection_string = args.connect
connection_baudrate = args.baudrate
vision_position_estimate_msg_hz = args.vision_position_estimate_msg_hz
vision_position_delta_msg_hz = args.vision_position_delta_msg_hz
vision_speed_estimate_msg_hz = args.vision_speed_estimate_msg_hz
scale_calib_enable = args.scale_calib_enable
camera_orientation = args.camera_orientation
debug_enable = args.debug_enable

# Using default values if no specified inputs
if not connection_string:
    connection_string = connection_string_default
    print("INFO: Using default connection_string", connection_string)
else:
    print("INFO: Using connection_string", connection_string)

if not connection_baudrate:
    connection_baudrate = connection_baudrate_default
    print("INFO: Using default connection_baudrate",
connection_baudrate)
else:
    print("INFO: Using connection_baudrate", connection_baudrate)

if not vision_position_estimate_msg_hz:
    vision_position_estimate_msg_hz = vision_position_estimate_msg_hz_default
    print("INFO: Using default vision_position_estimate_msg_hz",
vision_position_estimate_msg_hz)
```

```
else:
    print("INFO: Using vision_position_estimate_msg_hz",
vision_position_estimate_msg_hz)

if not vision_position_delta_msg_hz:
    vision_position_delta_msg_hz = vision_position_delta_msg_hz_default
    print("INFO: Using default vision_position_delta_msg_hz",
vision_position_delta_msg_hz)
else:
    print("INFO: Using vision_position_delta_msg_hz",
vision_position_delta_msg_hz)

if not vision_speed_estimate_msg_hz:
    vision_speed_estimate_msg_hz = vision_speed_estimate_msg_hz_default
    print("INFO: Using default vision_speed_estimate_msg_hz",
vision_speed_estimate_msg_hz)
else:
    print("INFO: Using vision_speed_estimate_msg_hz",
vision_speed_estimate_msg_hz)

if body_offset_enabled == 1:
    print("INFO: Using camera position offset: Enabled, x y z is",
body_offset_x, body_offset_y, body_offset_z)
else:
    print("INFO: Using camera position offset: Disabled")



#Camera Orientation
    H_aeroRef_T265Ref   = np.array([[0,0,-1,0],[1,0,0,0],[0,-1,0,0],[0,0,0,1]])
    H_T265body_aeroBody = np.linalg.inv(H_aeroRef_T265Ref)


if not debug_enable:
    debug_enable = 0
else:
    debug_enable = 1
    np.set_printoptions(precision=4, suppress=True) # Format output on
terminal
```

```
    print("INFO: Debug messages enabled.")



# https://mavlink.io/en/messages/common.html#VISION_POSITION_ESTIMATE
def send_vision_position_estimate_message():
    global is_vehicle_connected, current_time_us, H_aeroRef_aeroBody,
reset_counter
    with lock:
        if is_vehicle_connected == True and H_aeroRef_aeroBody is not
None:
            # Setup angle data
            rpy_rad = np.array( tf.euler_from_matrix(H_aeroRef_aeroBody,
'sxyz'))

            # Setup covariance data, which is the upper
right triangle of the covariance matrix, see here:
https://files.gitter.im/ArduPilot/VisionProjects/1DpU/image.png
            # Attemp #01: following this formula
https://github.com/IntelRealSense/realsense-ros/blob/development/realsense2_camera/src/
            cov_pose    = linear_accel_cov * pow(10, 3 -
int(data.tracker_confidence))
            cov_twist   = angular_vel_cov  * pow(10, 1 -
int(data.tracker_confidence))
            covariance  = np.array([cov_pose, 0, 0, 0, 0, 0,
                                    cov_pose, 0, 0, 0, 0,
                                       cov_pose, 0, 0, 0,
                                         cov_twist, 0, 0,
                                            cov_twist, 0,
                                               cov_twist])

            # Setup the message to be sent
            msg = vehicle.message_factory.vision_position_estimate_encode(
                current_time_us,            # us Timestamp (UNIX time or
time since system boot)
                H_aeroRef_aeroBody[0][3],   # Global X position
                H_aeroRef_aeroBody[1][3],   # Global Y position
                H_aeroRef_aeroBody[2][3],   # Global Z position
                rpy_rad[0],                # Roll angle
```

```
                rpy_rad[1],               # Pitch angle
                rpy_rad[2],               # Yaw angle
                covariance,                # Row-major representation
of pose 6x6 cross-covariance matrix
                reset_counter              # Estimate reset counter.
Increment every time pose estimate jumps.
            )
            vehicle.send_mavlink(msg)
            vehicle.flush()

# https://mavlink.io/en/messages/ardupilotmega.html#VISION_POSITION_DELTA
def send_vision_position_delta_message():
    global is_vehicle_connected, current_time_us,
current_confidence_level, H_aeroRef_aeroBody
    with lock:
        if is_vehicle_connected == True and H_aeroRef_aeroBody is not
None:
            # Calculate the deltas in position, attitude and time from
the previous to current orientation
            H_aeroRef_PrevAeroBody      = send_vision_position_delta_message.H_aeroRef_
            H_PrevAeroBody_CurrAeroBody = (np.linalg.inv(H_aeroRef_PrevAeroBody)).dot(H_

            delta_time_us    = current_time_us -
send_vision_position_delta_message.prev_time_us
            delta_position_m = [H_PrevAeroBody_CurrAeroBody[0][3],
H_PrevAeroBody_CurrAeroBody[1][3], H_PrevAeroBody_CurrAeroBody[2][3]]
            delta_angle_rad  = np.array( tf.euler_from_matrix(H_PrevAeroBody_CurrAeroBo
'sxyz'))

            # Send the message
            msg = vehicle.message_factory.vision_position_delta_encode(
                current_time_us,    # us: Timestamp (UNIX time or time
since system boot)
                delta_time_us,    # us: Time since last reported camera
frame
                delta_angle_rad,    # float[3] in radian: Defines a
rotation vector in body frame that rotates the vehicle from the previous
to the current orientation
                delta_position_m,   # float[3] in m: Change in position
```

```
from previous to current frame rotated into body frame (0=forward,
1=right, 2=down)
                current_confidence_level # Normalized confidence value
from 0 to 100.
            )
            vehicle.send_mavlink(msg)
            vehicle.flush()

            # Save static variables
            send_vision_position_delta_message.H_aeroRef_PrevAeroBody =
H_aeroRef_aeroBody
            send_vision_position_delta_message.prev_time_us =
current_time_us

# https://mavlink.io/en/messages/common.html#VISION_SPEED_ESTIMATE
def send_vision_speed_estimate_message():
    global is_vehicle_connected, current_time_us, V_aeroRef_aeroBody,
reset_counter
    with lock:
        if is_vehicle_connected == True and V_aeroRef_aeroBody is not
None:


            cov_pose    = linear_accel_cov * pow(10, 3 -
int(data.tracker_confidence))
            covariance  = np.array([cov_pose,   0,          0,
                                    0,          cov_pose,   0,
                                    0,          0,          cov_pose])

            # Setup the message to be sent
            msg = vehicle.message_factory.vision_speed_estimate_encode(
                current_time_us,            # us Timestamp (UNIX time or
time since system boot)
                V_aeroRef_aeroBody[0][3],   # Global X speed
                V_aeroRef_aeroBody[1][3],   # Global Y speed
                V_aeroRef_aeroBody[2][3],   # Global Z speed
                covariance,                 # covariance
                reset_counter               # Estimate reset counter.
Increment every time pose estimate jumps.
```

```
            )
            vehicle.send_mavlink(msg)
            vehicle.flush()

# Update the changes of confidence level on GCS and terminal
def update_tracking_confidence_to_gcs():
    if update_tracking_confidence_to_gcs.prev_confidence_level !=
data.tracker_confidence:
        confidence_status_string = 'Tracking confidence: ' +
pose_data_confidence_level[data.tracker_confidence]
        send_msg_to_gcs(confidence_status_string)
        update_tracking_confidence_to_gcs.prev_confidence_level =
data.tracker_confidence

# https://mavlink.io/en/messages/common.html#STATUSTEXT
def send_msg_to_gcs(text_to_be_sent):
    # MAV_SEVERITY: 0=EMERGENCY 1=ALERT 2=CRITICAL 3=ERROR, 4=WARNING,
5=NOTICE, 6=INFO, 7=DEBUG, 8=ENUM_END
    # Defined here: https://mavlink.io/en/messages/common.html#MAV_SEVERITY
    # MAV_SEVERITY = 3 will let the message be displayed on Mission
Planner HUD, but 6 is ok for QGroundControl
    if is_vehicle_connected == True:
        text_msg = 'T265: ' + text_to_be_sent
        status_msg = vehicle.message_factory.statustext_encode(
            6,                      # MAV_SEVERITY
            text_msg.encode()     # max size is char[50]
        )
        vehicle.send_mavlink(status_msg)
        vehicle.flush()
        print("INFO: " + text_to_be_sent)
    else:
        print("INFO: Vehicle not connected. Cannot send text message to
Ground Control Station (GCS)")

# Send a mavlink SET_GPS_GLOBAL_ORIGIN message (http://mavlink.org/messages/common#SET_
which allows us to use local position information without a GPS.
def set_default_global_origin():
    if is_vehicle_connected == True:
        msg = vehicle.message_factory.set_gps_global_origin_encode(
```

```
            int(vehicle._master.source_system),
            home_lat,
            home_lon,
            home_alt
        )

        vehicle.send_mavlink(msg)
        vehicle.flush()

# Send a mavlink SET_HOME_POSITION message (http://mavlink.org/messages/common#SET_HOME_
which allows us to use local position information without a GPS.
def set_default_home_position():
    if is_vehicle_connected == True:
        x = 0
        y = 0
        z = 0
        q = [1, 0, 0, 0]   # w x y z

        approach_x = 0
        approach_y = 0
        approach_z = 1

        msg = vehicle.message_factory.set_home_position_encode(
            int(vehicle._master.source_system),
            home_lat,
            home_lon,
            home_alt,
            x,
            y,
            z,
            q,
            approach_x,
            approach_y,
            approach_z
        )

        vehicle.send_mavlink(msg)
        vehicle.flush()
```

```
# Request a timesync update from the flight controller, for future work.
# TODO: Inspect the usage of timesync_update
def update_timesync(ts=0, tc=0):
    if ts == 0:
        ts = int(round(time.time() * 1000))
    msg = vehicle.message_factory.timesync_encode(
        tc,     # tc1
        ts      # ts1
    )
    vehicle.send_mavlink(msg)
    vehicle.flush()

# Listen to attitude data to acquire heading when compass data is
enabled
def att_msg_callback(self, attr_name, value):
    global heading_north_yaw
    if heading_north_yaw is None:
        heading_north_yaw = value.yaw
        print("INFO: Received first ATTITUDE message with heading yaw",
heading_north_yaw * 180 / m.pi, "degrees")
    else:
        heading_north_yaw = value.yaw
        print("INFO: Received ATTITUDE message with heading yaw",
heading_north_yaw * 180 / m.pi, "degrees")

def vehicle_connect():
    global vehicle, is_vehicle_connected

    try:
        vehicle = connect(connection_string, wait_ready = True, baud =
connection_baudrate, source_system = 1)
    except:
        print('Connection error! Retrying...')
        sleep(1)

    if vehicle == None:
        is_vehicle_connected = False
        return False
    else:
```

```
            is_vehicle_connected = True
            return True


    def realsense_notification_callback(notif):
        global reset_counter
        print("INFO: T265 event: " + notif)
        if notif.get_category() is rs.notification_category.pose_relocalization:
            reset_counter += 1
            send_msg_to_gcs('Relocalization detected')

    def realsense_connect():
        global pipe, pose_sensor

        # Declare RealSense pipeline, encapsulating the actual device and
    sensors
        pipe = rs.pipeline()

        # Build config object before requesting data
        cfg = rs.config()

        # Enable the stream we are interested in
        cfg.enable_stream(rs.stream.pose) # Positional data

        # Configure callback for relocalization event
        device = cfg.resolve(pipe).get_device()
        pose_sensor = device.first_pose_sensor()
        pose_sensor.set_notifications_callback(realsense_notification_callback)

        # Start streaming with requested config
        pipe.start(cfg)

    # Monitor user input from the terminal and perform action accordingly
    def user_input_monitor():
        global scale_factor
        while True:
            # Special case: updating scale
            if scale_calib_enable == True:
                scale_factor = float(input("INFO: Type in new scale as float
```

```
number\n"))
            print("INFO: New scale is ", scale_factor)

        if enable_auto_set_ekf_home:
            send_msg_to_gcs('Set EKF home with default GPS location')
            set_default_global_origin()
            set_default_home_position()
            time.sleep(1) # Wait a short while for FCU to start working

        # Add new action here according to the key pressed.
        # Enter: Set EKF home when user press enter
        try:
            c = input()
            if c == "":
                send_msg__to_gcs('Set EKF home with default GPS
location')
                set_default_global_origin()
                set_default_home_position()
            else:
                print("Got keyboard input", c)
        except IOError: pass



print("INFO: Connecting to vehicle.")
while (not vehicle_connect()):
    pass
print("INFO: Vehicle connected.")

send_msg_to_gcs('Connecting to camera...')
realsense_connect()
send_msg_to_gcs('Camera connected.')

if compass_enabled == 1:
    # Listen to the attitude data in aeronautical frame
    vehicle.add_message_listener('ATTITUDE', att_msg_callback)

# Send MAVlink messages in the background at pre-determined frequencies
sched = BackgroundScheduler()
```

```
if enable_msg_vision_position_estimate:
    sched.add_job(send_vision_position_estimate_message, 'interval',
seconds = 1/vision_position_estimate_msg_hz)

if enable_msg_vision_position_delta:
    sched.add_job(send_vision_position_delta_message, 'interval',
seconds = 1/vision_position_delta_msg_hz)
    send_vision_position_delta_message.H_aeroRef_PrevAeroBody =
tf.quaternion_matrix([1,0,0,0])
    send_vision_position_delta_message.prev_time_us =
int(round(time.time() * 1000000))

if enable_msg_vision_speed_estimate:
    sched.add_job(send_vision_speed_estimate_message, 'interval',
seconds = 1/vision_speed_estimate_msg_hz)

if enable_update_tracking_confidence_to_gcs:
    sched.add_job(update_tracking_confidence_to_gcs, 'interval', seconds
= 1/update_tracking_confidence_to_gcs_hz_default)
    update_tracking_confidence_to_gcs.prev_confidence_level = -1

# A separate thread to monitor user input
user_keyboard_input_thread = threading.Thread(target=user_input_monitor)
user_keyboard_input_thread.daemon = True
user_keyboard_input_thread.start()

sched.start()

if compass_enabled == 1:
    time.sleep(1) # Wait a short while for yaw to be correctly initiated

send_msg_to_gcs('Sending vision messages to FCU')

print("INFO: Press Enter to set EKF home at default location")

try:
    while True:
        # Monitor last_heartbeat to reconnect in case of lost connection
```

```
        if vehicle.last_heartbeat > connection_timeout_sec_default:
            is_vehicle_connected = False
            print("WARNING: CONNECTION LOST. Last hearbeat was %f sec
ago."% vehicle.last_heartbeat)
            print("WARNING: Attempting to reconnect ...")
            vehicle_connect()
            continue

        # Wait for the next set of frames from the camera
        frames = pipe.wait_for_frames()

        # Fetch pose frame
        pose = frames.get_pose_frame()

        # Process data
        if pose:
            with lock:
                # Store the timestamp for MAVLink messages
                current_time_us = int(round(time.time() * 1000000))

                # Pose data consists of translation and rotation
                data = pose.get_pose_data()


                # In transformations, Quaternions w+ix+jy+kz are
represented as [w, x, y, z]!
                H_T265Ref_T265body = tf.quaternion_matrix([data.rotation.w,
data.rotation.x, data.rotation.y, data.rotation.z])
                H_T265Ref_T265body[0][3] = data.translation.x *
scale_factor
                H_T265Ref_T265body[1][3] = data.translation.y *
scale_factor
                H_T265Ref_T265body[2][3] = data.translation.z *
scale_factor

                # Transform to NED coordinates (body and inertial
reference frame)
                H_aeroRef_aeroBody = H_aeroRef_T265Ref.dot(
H_T265Ref_T265body.dot( H_T265body_aeroBody))
```

```
                # Calculate GLOBAL XYZ speed (speed from T265 is already
GLOBAL)
                V_aeroRef_aeroBody = tf.quaternion_matrix([1,0,0,0])
                V_aeroRef_aeroBody[0][3] = data.velocity.x
                V_aeroRef_aeroBody[1][3] = data.velocity.y
                V_aeroRef_aeroBody[2][3] = data.velocity.z
                V_aeroRef_aeroBody = H_aeroRef_T265Ref.dot(V_aeroRef_aeroBody)

                # Check for pose jump and increment reset_counter
                if prev_data != None:
                    delta_translation = [data.translation.x -
prev_data.translation.x, data.translation.y - prev_data.translation.y,
data.translation.z - prev_data.translation.z]
                    position_displacement = np.linalg.norm(delta_translation)


                    jump_threshold = 0.1 # in meters, from trials and
errors, should be relative to how frequent is the position data obtained
(200Hz for the T265)
                    if (position_displacement > jump_threshold):
                        send_msg_to_gcs('Pose jump detected')
                        print("Position jumped by: ",
position_displacement)
                        reset_counter += 1

                prev_data = data

                # Offset
                if body_offset_enabled == 1:
                    H_body_camera = tf.euler_matrix(0, 0, 0, 'sxyz')
                    H_body_camera[0][3] = body_offset_x
                    H_body_camera[1][3] = body_offset_y
                    H_body_camera[2][3] = body_offset_z
                    H_camera_body = np.linalg.inv(H_body_camera)
                    H_aeroRef_aeroBody = H_body_camera.dot(H_aeroRef_aeroBody.dot(H_cam

except KeyboardInterrupt:
    send_msg_to_gcs('Closing the script...')
```

```
except:
    send_msg_to_gcs('ERROR IN SCRIPT')
    print("Unexpected error:", sys.exc_info()[0])

finally:
    pipe.stop()
    vehicle.close()
    print("INFO: Realsense pipeline and vehicle object closed.")
    sys.exit()
```