UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA DEL CAMINO ÓPTIMO EN MAPEO DE HABITACIÓN SIMULADA UTILIZANDO DRONES CON SONAR

TESIS PARA OPTAR AL GRADO DE MAGISTER EN CIENCIAS, MENCIÓN COMPUTACIÓN.

MEMORIA PARA OPTAR AL TITULO DE INGENIERO CIVIL EN COMPUTACIÓN

ENZO PIO DAVID CERFOGLI COPPA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER
PROFESOR GUÍA 2:
PABLO GUERRERO PÉREZ

MIEMBROS DE LA COMISIÓN:
JORGE ANDRES BAIER ARANDA
BENJAMIN BUSTOS CARDENAS
GONZALO NAVARRO BADINO

SANTIAGO DE CHILE
2020

Resumen

Los robots domésticos se enfrentan a diferentes y complicados ambientes, gracias a su producción en masa y aumento de su complejidad. Para poder aprovechar sus capacidades y poder interactuar con el ambiente y sus usuarios, se necesita un mapa, que contenga información contextual y que les permita navegar en su entorno.

En este trabajo se busca generar un mapa de tres dimensiones de habitaciones. Dada la complejidad de esto se hicieron dos simplificaciones: la primera, se usaron mapas sencillos generados artificialmente; la segunda, considera que el robot solo tiene que captar con su sensor un punto del mapa, para considerarlo como correcto, en vez de tomar mediciones y editar el mapa acorde a estas mediciones. Estas restricciones se tomaron para enfocarse en probar los métodos de optimización y, además, probar si una simplificación de este problema puede ser resuelto.

Para poder revisar el mapa se diseñó e implementó un algoritmo, que permite revisar todos los puntos pertenecientes a una nube de puntos, que es como se definió el mapa. Para esto se utilizaron cuatro módulos encargados de: Odometría, control y planeación de trayectoria, determinación de pose optima, contención y manejo del mapa. Con estos módulos se efectuó la comparación de los métodos de optimización de SciPy, en tres diferentes mapas. Como métrica de eficiencia se usó el porcentaje de puntos vistos, sobre la cantidad de puntos totales del mapa.

Solamente tres métodos de optimización, Powell, Neldel-Mead y COBYLA, pertenecientes a la librería SciPy, lograron revisar todos los mapas. En particular el método Powell se demoró 44 ciclos en revisar 75 % de los puntos del mapa cuadrado, el método Neldel-Mead tomó 68 ciclos y al método COBYLA le tomó 61 ciclos en las mismas condiciones. Este es un ejemplo representativo de los resultados obtenidos, donde Powell logró superar en menor cantidad de ciclos al resto de los métodos en los tres mapas.

Es interesante notar, que el método Powell que tiene menor complejidad, sea el que tiene menor cantidad de ciclos. Con esto se demuestra que, a pesar de las simplificaciones del problema, se puede lograr revisar todos los puntos de la nube que componen el mapa. Esto confirma la hipótesis que existe un método de optimización que logra generar un mapa de tres dimensiones.

Agradecimientos

A mis profesores guías, Nancy Hitschfeld y Pablo Guerrero por su continua orientación y conocimientos durante la tesis.

A mi familia por su constante apoyo durante toda mi educación.

A mi pareja por su soporte y compañía durante el transcurso carrera.

A mis compañeros por su ayuda durante las noches de estudio.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

La exploración del ambiente que nos rodea ha sido un tema de interés desde hace mucho tiempo. Nos permite entender dónde estamos, dónde podemos ir y comunicar nuestros objetivos. Antes de los inventos tecnológicos, los humanos estaban restringidos a la descripción de puntos de referencia en el ambiente, para saber dónde se encontraban, por lo tanto, su alcance máximo estaba limitado por la visualización de un punto de referencia conocido.

Con la aparición de diferentes inventos tecnológicos, mejoró la capacidad de detectar estos puntos importantes y de medir su distancia con respecto a los humanos. Algunos ejemplos destacables son:

- Astrolabio: Permitió medir los ángulos entre el observador y las constelaciones.
- Brújula: Permitió encontrar un punto de referencia absoluto, el norte magnético, sin importar del ciclo día-noche o nubosidad.
- Radio-faros: Permitió crear puntos de referencia artificiales absolutos con un gran rango de detección.
- GPS: Los satélites permitieron crear un mapa fotográfico del mundo y además sirven como punto de referencia para determinar la posición del observador.

Con la aparición de los robots domésticos, surgió la necesidad de crear un mapa que esté disponible para ellos, de tal forma que puedan realizar tareas complejas. Con un mapa, se pueden marcar zonas prohibidas, donde el robot no pueda pasar, protegiéndolo de áreas peligrosas, como escaleras. También puede utilizar la información contenida en el mapa, para cumplir su objetivo.

Actualmente los robots domésticos son pequeños y con ruedas, pero cada vez se hacen más baratos y complejos, existiendo incluso aquellos con apariencia humana y otros con la capacidad de flotar. Considerando esto, resulta necesario la generación de mapas que puedan acomodarse a sus necesidades y capacidades, como por ejemplo el mapeo tridimensional de espacios cerrados.

1.1. Motivación

En la actualidad existen distintas aplicaciones que realizan mapeo de espacios y a la vez, distintos mapas disponibles. Un ejemplo de esto es *Google Maps*, que es un servicio de aplicaciones de mapa que incluye imágenes de mapa desplazables, fotografía de satélites y rutas entre diferentes ubicaciones.

En la actualidad las aspiradoras Roomba desde la versión 900 tienen un sistema de mapeo en dos dimensiones, con este mapa puede eficientemente limpiar una habitación y se pueden marcar zonas prohibidas.

En ROS (Robot Operative System) existe una librería llamada gmapping[1] que permite mapeo de dos dimensiones usando un robot móvil, un telémetro láser y método de filtro de partículas, pero esta biblioteca no tiene control para un dron ni mapeo en tres dimensiones.

Esto nos lleva a preguntar: ¿Es posible generar un método iterativo que encuentre una secuencia óptima de poses del robot para construir mapas en tres dimensiones de habitaciones?

1.2. Hipótesis

Es posible generar un mapa 3D en tiempo finito, de forma óptima, utilizando los métodos de optimización disponibles en la literatura

1.3. Objetivos

Objetivo General: Diseñar y desarrollar un método de planificación de trayectoria de un robot móvil, para construir de forma eficiente un modelo tridimensional del entorno.

Los objetivos específicos son:

- Definir e implementar una forma de representar el mapa y su incertidumbre[2].
- Definir métrica de optimización que busca medir el progreso del mapeo en la habitación.
- Diseñar e implementar un sistema de seguimiento de trayectoria.
- Diseñar e implementar un algoritmo iterativo, que determine la siguiente pose[3] óptima en función del mapa actual.
- Diseñar y realizar experimentos para comparar métodos de optimización.

[1] https://openslam-org.github.io/gmapping.html
[2] Medición de la probabilidad de que el mapa sea fidedigno
[3] Orientación y posición ver en conceptos básicos

1.4. Metodología

Para desarrollar esta tesis se definen los siguientes pasos:

1. Desarrollo del algoritmo.
   (a) Definir librerías y versiones existentes para:
      i. Sistema de localización.
      ii. Control y seguimiento de trayectoria.
      iii. Métodos de optimización.
      iv. Triangulación de Delaunay.
   (b) Diseñar e implementar algoritmos para:
      i. Sistema de localización.
      ii. Control y seguimiento de trayectoria.
      iii. Cálculo de pose óptima.
      iv. Almacenamiento e interacción de mapa.
2. Validación del algoritmo en un simulador.
   (a) Definir escenario.
   (b) Medir y comparar los métodos de optimización en el simulador.

El proyecto se realizó en cinco nodos[4] de ROS en Python. Los *input* cruciales de los nodos son:

- La pose estimada.
- El mapa representado por una nube de puntos.

Para el control y seguimiento de trayectoria, los *inputs* del nodo se usan para determinar la pose que disminuya la incertidumbre del mapa, luego el dron[5] se mueve hasta esta nueva pose, donde se repite este proceso definiendo el algoritmo iterativo. El algoritmo termina cuando la métrica de optimización cumpla el criterio establecido y envía la señal de aterrizaje al dron.

Para el control y representación del universo, se usa una representación de mallas geométricas tridimensionales, donde cada punto está sujeto a incertidumbre. Se busca elegir una representación que permita generar un algoritmo iterativo eficiente para incorporar las observaciones, actualizar la malla y su representación de incertidumbre.

Para el desarrollo del mapa, la métrica de optimización es orientada a reducir la incertidumbre. Se evalúa la métrica el porcentaje de puntos vistos contra los puntos totales. Se considera para la elección de la siguiente pose del dron la pose óptima de acuerdo con la métrica anteriormente mencionada.

[4] los nodos son procesos que realizan las rutinas en un robot en ROS
[5] Una aeronave no tripulada, que puede funcionar con ciertos de grados de autonomía

1.5. Aporte

Los aportes de la tesis son:

- El nuevo algoritmo: El uso de mallas geométricas es nuevo en el contexto de algoritmo de localización y mapeado simultáneo (SLAM), por lo tanto, el algoritmo y la métrica de optimización es un aporte.
- Casos de prueba: No existen casos de prueba y métricas disponibles en la comunidad, para ello se dejarán disponibles, en el repositorio de la tesis [6].

1.6. Contenido de la tesis

Introducción Se presentan los aspectos generales del trabajo, la motivación, los objetivo generales y específicos, la metodología y el aporte.

Antecedentes Se presentan con mayor detalle los temas a tratar, los conceptos referentes al problema planteado, los métodos existentes que pueden ayudar a resolver dichos problemas, los métodos de optimización, herramientas de localización y revisión del estado del arte.

Análisis y diseño Se presenta el detalle del problema, los requerimientos funcionales de los módulos de la solución, la arquitectura de la solución, y el diseño de los módulos, sus submódulos y funciones importantes.

Implementación Se presenta una descripción de código de cada módulo y las funcionalidades importantes de estos.

Medición y análisis Se presenta los mapas diseñados donde se prueban los métodos de optimización, los resultados obtenidos por cada mapa y por cada método de optimización

Conclusión y trabajo futuro Se presentan las conclusiones obtenidas a partir del trabajo realizado, se revisa compara los métodos de optimización, y se analiza el posible trabajo futuro que se puede llevar a cabo al mejorar la solución.

Bibliografía Se muestran las fuentes de información bibliográfica utilizadas tanto en el estado del arte, como en el diseño e implementación y la referencia de imágenes obtenidas en internet.

[6] https://bitbucket.org/enzo_cerfogli/dron_path/src

Anexo Se presentan los gráficos de los métodos que no lograron completar todo el mapa.

Capítulo 2

Antecedentes

En este capítulo se describen los conceptos básicos y herramientas que anteceden al desarrollo de esta tesis de grado. Estos temas son requisitos básicos para poder comprender las razones y justificaciones del desarrollo de esta aplicación, junto con las definiciones utilizadas y desarrolladas en el diseño.

Primero se describen herramientas importantes para el desarrollo de software en el área de robótica, como ROS, y otros conceptos requeridos para entender la tesis. Luego, se exponen los métodos de optimización que se utilizan y finalmente, se describe la triangulación de Delaunay que se usa para definir una vecindad entre los puntos.

2.1. Conceptos básicos

A continuación, se explican algunos conceptos usados en la tesis

- Dron: Se denomina dron a una aeronave no tripulada, que puede funcionar con ciertos grados de autonomía, desde control remoto de una persona o automático por computadores a bordo de la aeronave, como por ejemplo el Parrot Ar.Drone.
- Parrot AR.Drone 2.0: Es un modelo de dron usado por civiles para fines recreativos, funciona propulsado por cuatro motores eléctricos en configuración de cuadricóptero como se ve en la figura 2.1. Posee un microprocesador y una serie de sensores, que incluye 2 cámaras, un sonar y un adaptador Wi-Fi, lo que le permite conectarse a dispositivos. Fue vendido por la empresa Parrot[1] y se encuentra descontinuado.

[1] https://www.parrot.com/global/drones/parrot-ardrone-20-elite-edition

Figura 2.1: AR.Drone 2.0 [3]

- Odometría: Es el uso de datos provenientes de sensores internos para estimar la posición del dron (unidad de medición de inercia, contadores de revoluciones del motor). Dado que los sensores tienen un error, éste se acumula con el tiempo, por lo cual no se puede confiar al largo plazo en la odometría sola.
- Inertial Measurement Unit: El componente unidad de medición inercial (*inertial measurement unit* IMU) permite medir orientación, velocidad y la fuerza gravitacional, gracias a giroscopios y magnetómetro.
- Sonar: Es una técnica que usa ondas de sonido y su propagación y reflejo para medir distancias a los objetos, mediante el efecto Doppler.
- Pose: En robótica la pose se refiere a la posición y orientación del robot en algún sistema de referencia. Para la tesis como posición se considera los datos en un espacio euclidiano (x,y,z) y para la orientación se usaron los ángulos de Euler o de navegación que son 3:
  - Cabeceo o *pitch* que es el ángulo con respecto al eje x.
  - Alabeo o *roll* que es el ángulo con respecto al eje y.
  - Guiñada o *yaw* que es el ángulo con respecto al eje z.

  En la figura 2.2 se muestran los ángulos.

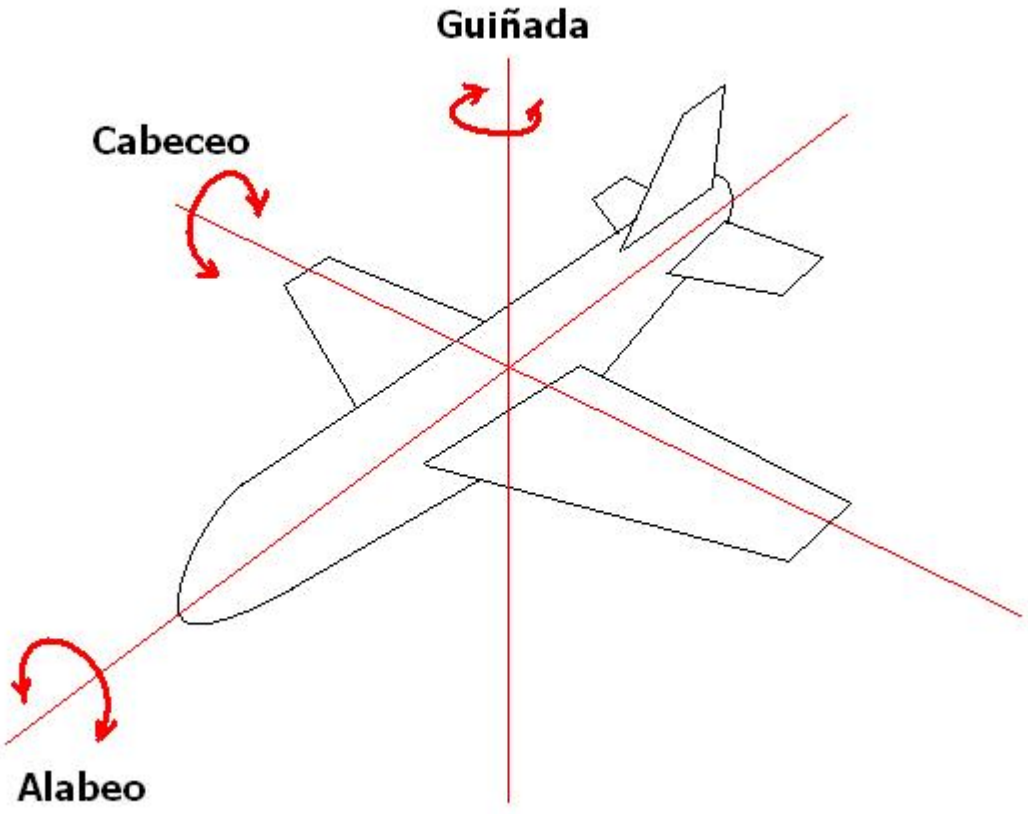


Figura 2.2: Ángulos de Euler [15]

- Código de respuesta rápida: Un código de repuesta rápida (Quick Response o QR) es un código de barra bidimensional que puede almacenar datos codificados. Dependiendo del formato puede guardar 4293 caracteres, además de tener métodos de verificación y medidas para recuperar la información. Se muestra un ejemplo en la figura 2.3.

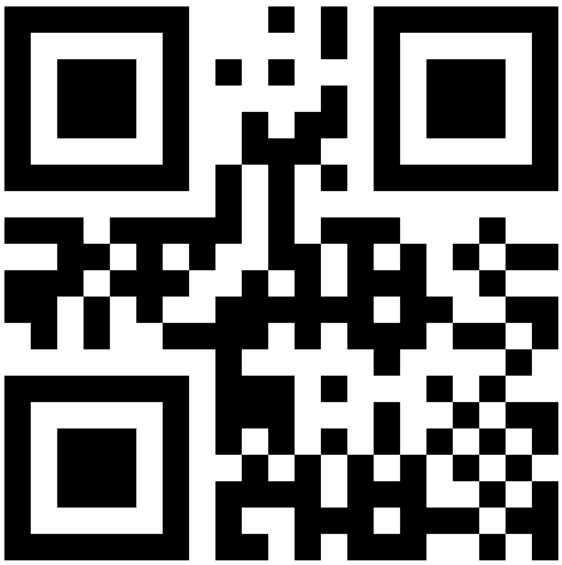


Figura 2.3: Un código de respuesta rápida que contiene: (0.0.0)

- Nube de puntos: Se define nube de puntos a un conjunto de vértices que representan una superficie de un objeto en tres dimensiones.
- Inecuación del cono: El área contenida dentro de un cono, con vértice en el punto $(0; 0)$ y eje alineado con el eje x, se puede definir por la inecuación:

  $$F(x; y; z) = (x^2 + y^2) * (cos\theta)^2 - z^2 * (sin\theta)^2$$

  Con las restricciones $\{F(x, y, z) < 0\ ,\ 0 < z < h\}$ donde h es la altura del cono y 2θ es ángulo de apertura. Se usa esta inecuación por su simplicidad de comprensión y para determinar qué puntos están dentro del cono de visión.
- Ángulo entre vectores: Para determinar el ángulo entre vectores se puede usar la fórmula: coseno del ángulo entre vectores equivale al producto escalar de dos vectores dividido en el producto de módulos de estos vectores. Como se ve en la forma escrita: $\theta = arccos(A \cdot B/|A| * |B|)$. En esta tesis A es la normal estimada usando la vecindad de puntos en la malla y B es la orientación del dron.
- Mapeo y localización simultáneos: Para poder hacer un mapa, se hacen mediciones del entorno desde el robot con la finalidad de guardarlas y luego movemos el robot a algún punto para realizar otras mediciones y guardarlas nuevamente. Sin embargo, este movimiento es impreciso y con los continuos movimientos del robot este error se acumula. Usualmente se compara la posición del robot en el mapa para compensar este error, por lo cual esto se vuelve un problema del huevo y la gallina, se necesita un mapa para saber la posición y se necesita la posición para generar el mapa. A este problema se le llama SLAM (por sus siglas en ingles Simultaneous Localization and Mapping).
- Incerteza: La incerteza se caracteriza por ser la dispersión de los valores que pueden atribuirse razonablemente a dicho resultado de medición. Hay que considerar también que incerteza se puede propagar.

2.2. Herramientas disponibles

En esta sección se presentan las herramientas disponibles para desarrollar el algoritmo, tales como ROS y los métodos de optimización.

2.2.1. Sistema Operativo Robótico

El Sistema Operativo para robots (Robot Operating System o ROS) es un *framework* de código abierto para la creación de software para robots, actualmente siendo desarrollado por Open Source Robotics Fundation (OSRF). Este *framework* actúa como meta-sistema operativo, como plataforma de robots heterogénea, proveyendo abstracción del hardware, control de bajo nivel, implementación de funciones comúnmente usadas, manejo de mensajes entre procesos y manejo de librerías. ROS es una colección de herramientas, librerías y convenciones que apuntan a simplificar el desarrollo de software para robots. ROS está estructurado como una red de pares de procesos llamados nodos ROS. Estos nodos pueden ser ejecutados en diferentes máquinas y están conectados usando la infraestructura de comunicación provisto por ROS, incluyendo *streaming* asincrónica de datos y el guardado de datos como parámetros del servidor. Hay seis elementos en esta arquitectura:

- Nodos: los nodos son procesos que realizan las rutinas en un robot. ROS está diseñado de forma modular; el sistema de control de robot está compuesto por varios nodos. Por ejemplo, un nodo controla el telémetro láser, otro nodo controla los motores de las ruedas, un tercer nodo realiza la localización, un cuarto nodo realiza el planeo de camino, un quinto nodo provee una vista gráfica del sistema.
- Maestro: El nodo ROS maestro es un nodo que se encarga del registro de nombres de los nodos y maneja las consultas de los nodos en la red. Sin el nodo maestro, los nodos no podrían encontrarse entre ellos, intercambiar mensajes o invocar servicios.
- Parámetros del servidor[2]: Permite guardar información en un lugar central, se encuentra incluido en el nodo maestro.
- Mensaje: Los nodos se comunican unos con otros pasando mensajes. Un mensaje es simplemente una estructura de datos compuesta de valores con tipo definidos. Tipos estándares primitivos (*integer, floating point, boolean*, etc) están disponibles, juntos con los arreglos de estos.
- Tópico: Un tópico es un nombre que es usado para identificar el contenido del mensaje. Los mensajes son guiados vía un sistema de transporte con estructura publicador/suscriptor. Un nodo que está interesado en ese tipo de datos se suscribe al tópico apropiado. Un nodo envía un mensaje publicándolo en un tópico. Puede haber múltiples publicadores y suscriptores concurrentes en un solo tópico y un nodo puede publicar o suscribirse en múltiples tópicos. El objetivo es desacoplar el productor del mensaje con el consumidor.
- Servicio: Para mensajes que necesitan respuesta se utilizan servicios, que tienen estructura Solicitud / Respuesta, que se definen como una estructura de dos de mensajes: uno para la solicitud y el otro mensaje para la respuesta. Un nodo provee un servicio bajo un nombre y un cliente usa el servicio enviando un mensaje de petición y espera una respuesta.

[2]

Se usará ROS como *framework* para desarrollar el *software*.

2.2.2. Gazebo

Gazebo es un simulador gráfico en tres dimensiones desarrollado por la *Open Source Robotics Foundation* (OSRF), que permite simular robots en ambientes externos e internos. Esto permite simular las mediciones de los sensores del robot y que los actuadores del robot afecten el ambiente. Incluye módulos que permiten modelar el ambiente, los robots, los sensores, los actuadores y objetos con los cuales los robots puedan interactuar (cubos, tasas, mesas, etc) [11]. Se usará Gazebo para simular las mediciones del robot.

2.2.3. Métodos de optimización de SciPy

En la biblioteca que funciona en Python de código abierto para matemática, ciencias e ingeniería llamada SciPy[3], se encuentran disponibles métodos que buscan minimizar una función objetiva sobre varias variables, se clasifican dependiendo si existen restricciones o limites en los valores posibles.

2.2.3.1. Minimización sin restricciones

- Nelder-Mead. El método Nelder-Mead se basa en un símplex [4] que recorre una pendiente y toma una serie de pasos. En la mayoría de los pasos se mueve el punto del símplex donde la función es mayor ("punto más alto") a través de la cara opuesta del símplex a un punto más bajo, estos pasos son llamadas reflexiones, y son usados para conservar el volumen del símplex (y esto mantiene la no-degeneración [5]). Cuando el método puede, se expande el símplex en una u otra dirección para tomar grandes pasos. Cuando llega a un "valle", el método se contrae en la dirección transversa y trata de rezumar hacia abajo en el valle. Si hay una situación donde el símplex está tratando de "pasar a través del ojo de una aguja", se contrae a sí misma en todas las direcciones, traccionándose a sí mismo al punto más bajo [14].
- Powell. La idea general del método de optimización de Powell es buscar el mínimo de la función en cada vector canónico, luego usar el vector desplazado (la resta entre el punto inicial del paso anterior de optimización y el punto resultado del paso anterior) como vector de dirección para buscar el nuevo mínimo.[14]

  El ciclo de minimización es:

  1. Determinar punto inicial del mínimo sugerido (x_0 que puede ser aleatorio)
  2. Iterando por todo el conjunto de base canónica: por cada componente se busca el mínimo de la función. (x_1)

[3] https://www.scipy.org/

[4] En geometría un símplex o n-símplex es análogo a un triángulo en n-dimensiones

[5] la no-degeneración es que todos los lados del símplex y sus ángulos sean mayor a cero

3. Recalcular el vector de dirección a la resta entre el punto inicial y el punto calculado en la iteración anterior. ($P_1 = x_1 - x_0$)
4. Buscar el mínimo en la nueva dirección (P_1)
5. Repetir paso 2 y 3, hasta que la diferencia sea menor a un número establecido entre mínimos.

El método usado en la librería *SciPy* incluye 2 modificaciones:

- Si el procedimiento es en forma cuadrática una dirección conjugada[6] es elegida.
- En el tercer paso si el nuevo vector de direcciones no cumple una condición no es usado para buscar mínimo.

- Método del gradiente conjugado (CG). El método de gradientes conjugado sigue la misma lógica del método de Powell, la diferencia es cómo calcula el vector de dirección y el valor que multiplica a este vector, aprovechando la derivada en el punto.

  El método está pensado para minimizar una función cuadrática, usando la derivada para calcular el vector de dirección. La variante del método CG llamada Polak-Ribière utilizada en la librería calcula la variable β [beta] como [$\beta_n^{PR} = -\frac{\Delta x_n^T(\Delta x_n - \Delta x_{n-1})}{\Delta x_{n-1}^T \Delta x_{n-1}}$] que es el vector de dirección o búsqueda [14].
- Método Quasi-Newton de Broyden, Fletcher, Goldfarb, and Shanno (BFGS) Los métodos de Newton se basan en buscar las raíces de la derivada de la función a optimizar, para esto se usa la matriz Hessiana[7] Los métodos de Quasi-Newton estiman la matriz Hessiana dado que calcular el valor preciso es muy costoso en recursos de computación.

2.2.3.2. Minimización con restricciones de límites

- Método truncado de Newton (*Truncated Newton method* TNC), El método de TNC también llamado CG-Newton dado que se calcula la dirección de búsqueda aplicando el método CG a la ecuación de Newton. Notar, sin embargo, el método CG está diseñado para resolver sistemas definidos positivos, y la Hessiana puede tener valores/vectores propios negativos lejos de la solución [12].
- L-BFGS-B: Es una adaptación de BFGS para restricciones y grandes cantidades de variables.

2.2.3.3. Minimización con restricciones

- Optimización restringida por aproximación lineal (COBYLA). Es un algoritmo que construye una aproximación linear a la función objetivo y las funciones restrictivas por la interpolación de los vértices de un simplex y la región restringida en cada cambio a las variables. Entonces un nuevo vector de variables es calculado, el cual puede remplazar uno de los puntos actuales.[13]
- Programación cuadrática secuencial de cuadrados mínimos (*Sequential Least-Squares Quadratic Programming*, SLSQP). El algoritmo optimiza las sucesivas aproximaciones de segun-

[6]Dos vectores son conjugados si son ortogonales con respecto al producto interior.

[7]la matriz Hessiana de una función f de n variables, es la matriz cuadrada de n x n, de las segundas derivadas parciales.

do orden (cuadrático/cuadrados-mínimos) de la función objetiva a través de actualizaciones usando el algoritmo BFGS, con aproximaciones de primer orden en las restricciones. [5]

Se comparan las eficiencias de mapeo de la habitación entre estos métodos.

2.2.4. Filtro extendido de Kalman

El filtro Extendido de Kalman del 1960 se basa en el filtro de Kalman, que es un algoritmo donde usa una serie de medidas observadas en el tiempo, que contiene ruido estadístico y otras inexactitudes, y produce un estimado de la variable desconocida, usando la distribución de la probabilidad conjunta sobre cada ciclo. La diferencia es que el algoritmo extendido funciona sobre problemas no lineales, simplificándolos con expansión de serie de Taylor.

El algoritmo funciona en un proceso de dos fases. En fase de predicción, el filtro produce un estimado del estado de las variables y sus incertidumbres actualizando las del instante de tiempo anterior en función de las entradas recibidas en el intertanto. Una vez recibida la medición actual (típicamente corrompida con algún error incluyendo errores aleatorios de ruido), estos estimados son actualizados usando un peso promedio, con mayor peso a la estimación con mayor certeza. Este algoritmo es recursivo. Puede correr en tiempo real, usando solo: las mediciones presentes, el estado anterior y la matriz de incertidumbre.

El filtro se usa para crear una estimación de la pose, como menciona Thrun en [16]. Se usará un filtro extendido de Kalman para el sistema de localización.

2.2.5. Controlador Proporcional-Integral-Derivativo (PID)

El controlador Proporcional-Integral-Derivativo un mecanismo de control iterativo por retroalimentación. El controlador calcula continuamente el valor de error, que es la diferencia entre el valor deseado y el medido, y aplica una corrección basada en los términos: proporcional, integral y derivado.

Para explicar la funcionalidad del controlador PID analicemos una descripción gráfica de un PID en un sistema cerrado en la figura 2.4, la variable e representa el error analizado, que es la diferencia entre la salida deseada r y la salida actual y. Este error en la señal e (que depende del tiempo) es alimentado al controlador PID, que calcular la derivada y la integral de este error con respecto al tiempo. La señal control u que es enviada al proceso es igual a $u(t) = K_p\mathrm{e}(t) + K_\mathrm{i}\int_0^t \mathrm{e}(t)\mathrm{d}t + K_\mathrm{d}\frac{\mathrm{e}(t)}{\mathrm{d}t}$. Se describe la formula como: la constante proporcional K_p veces la magnitud del error más la constante integral K_i veces la integrar del error más la constante derivativa K_d veces la derivada del error. El control de la señal u es alimentada a la planta y el nuevo valor de salida y es obtenida. El control toma este nuevo error de la señal y rehace los cálculos de la entrada del control. Este proceso continúa mientras el controlador esté en efecto [4].

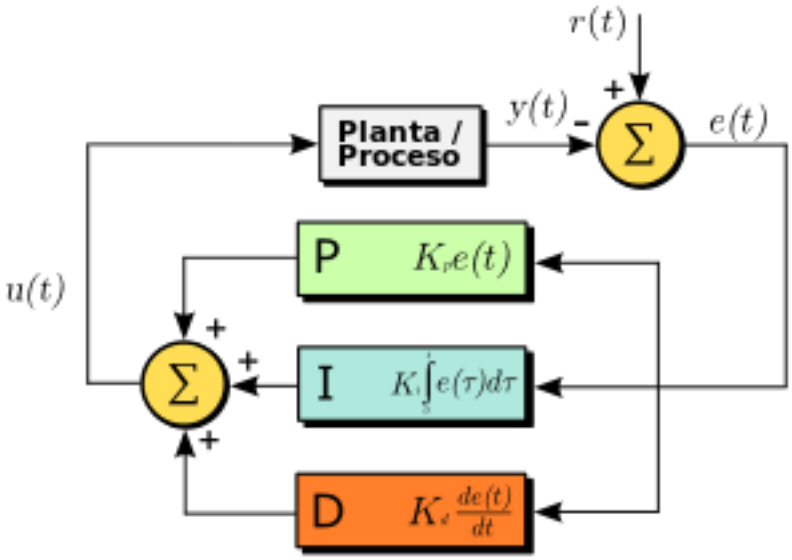


Figura 2.4: Modelo de un controlador PID [4]

Se usará el PID para el algoritmo de control y seguimiento de trayectoria.

2.2.6. Triangulación de Delaunay

Una malla triangular (o triangulación) es una especialización de una malla de polígonos, la cual es capaz de definir la forma de un objeto dividiéndolo en un conjunto de triángulos unidos por sus aristas, que forman una superficie.

La triangulación de Delaunay es una red de triángulos conexa y convexa que cumple el criterio de Delaunay, que dice: "Para un conjunto de puntos en dos dimensiones el criterio garantiza que el circuncírculo[8] asociado con cada triángulo no contenga ningún otro punto en su interior". Como ejemplo en la Figura 2.5 se observa que el vértice V3 no está en el circuncírculo de T1, ni V1 en el circuncírculo de T2, mientras que en la figura 2.6 el vértice V3 sí está en el circuncírculo de T1, y V1 está en el circuncírculo de T2.

[8]En geometría, la circunferencia circunscrita es la circunferencia que pasa por todos los vértices de un polígono y contiene completamente a dicha figura en su interior.

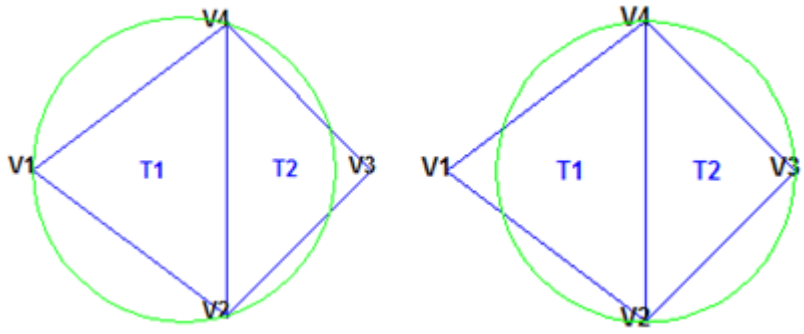


Figura 2.5: Ejemplo de una triangulación de Delaunay [1]

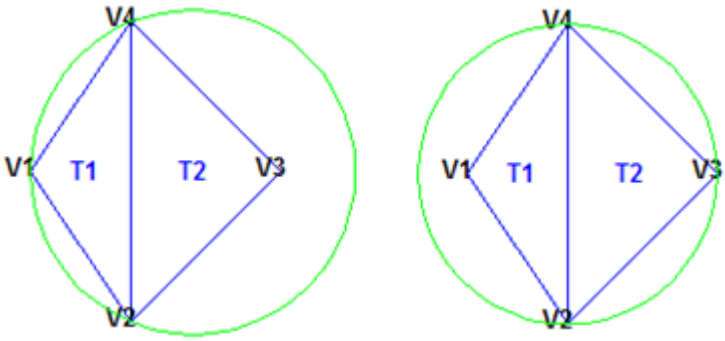


Figura 2.6: Ejemplo de una triangulación [1]

Se usará la triangulación de Delaunay para generar vecindad en el mapa.

2.3. Revisión del estado del arte

A pesar de que SLAM lleva años en la academia, e incluso se encuentra implementado en la industria, un ejemplo en robots aspiradora Roomba 910[9], estos están limitados a dos dimensiones incluso si se usa robots aéreos o humanoides. A continuación, se presenta a una revisión del estado del arte.

En el trabajo de R. Bajcsy et al. llamado "*Active perception*" [6], se propone un algoritmo que, dada una lista de poses del robot, se le asigna a cada pose una probabilidad de efectividad, es decir, qué pose contribuye con más información útil del entorno para decidir cuál elegir. Bajo este concepto, nace el *active* SLAM, donde el objetivo es mejorar los resultados del SLAM eligiendo la siguiente pose usando un algoritmo probabilístico. Desde la propuesta de R. Bajcsy han pasado más de 29 años desarrollándose diferentes métodos de SLAM, sin embargo, aún quedan muchos temas por mejorar como se ve en el trabajo de C. Cadena et al. [7], en el cual se plantean 3 de ellos:

[9]https://tienda.irobot.es/roomba-aspirador-robot-aspirador-roomba-980/R980040.html

1. ¿Cuándo detener el SLAM? En sí, el SLAM es un medio para lograr un objetivo, por lo cual no podemos utilizar un tiempo indefinido realizando el SLAM. Además, desde cierto momento, la información adicional puede ocasionar rendimientos decrecientes en el algoritmo o puede ser contradictoria a la información previamente obtenida.
2. ¿Qué pose puede dar más información? Considerando la alta incertidumbre de ciertas zonas del mapa, es difícil determinar la ganancia de información relevante en el mapa a partir de los puntos circundantes a estas zonas, por lo cual, sigue siendo necesario mejorar los algoritmos que permitan optimizar la determinación de la ganancia de información en cada punto y su consiguiente elección.
3. ¿Eventualmente, terminan los algoritmos? No existen demostraciones de que los algoritmos de SLAM terminen, por lo cual es importante desarrollar demostraciones matemáticas para tener la certeza de que éstas efectivamente terminan o no.

En el trabajo de Ayoung Kim et al. [10] los autores buscan resolver un subproblema del SLAM, llamado área de cobertura, donde se prioriza que el algoritmo sea rápido. Para esto Ayoung Kim et al. desarrollan un *framework* de recompensa llamado "*perception driven navigation*" que busca minimizar el área explorada redundante, contrapesando la opción de explorar contra revisitar, usando un algoritmo que incluye la incertidumbre acumulada de la pose y la eficiencia del área explorada contrastando esta última con el área por explorar y el costo de área revisitada.

En el trabajo de Gabriele Constante et al.[9] incluyen el uso de texturas de las imágenes obtenidas por el robot para reducir la incerteza del mapa, además de incluir también una preferencia por las zonas con grandes diferencias de texturas en la elección de la pose, aprovechando así esta característica. Por otro lado, Henry Carrillo et al. [8] usan las definiciones de entropía de Shannon y Rényi para determinar la incertidumbre compuesta del robot y el mapa para optimizar las elecciones de pose entre exploración y confirmación (volver al punto original para disminuir la incertidumbre de la pose).

Capítulo 3

Análisis del problema y diseño

En este capítulo se describen: Los requerimientos funcionales de la solución, la arquitectura del sistema propuesto, los módulos, sus objetivos, y las interacciones entre ellos.

3.1. Requerimientos funcionales del sistema

Se eligen habitaciones que tengan objetos que no interrumpan mayormente el vuelo del dron, pero que complejicen el mapa (por ejemplo: sillas, mesas, etc..) y un dron modelo Parrot AR.Drone 2.0 modificado con sonar. Se espera que el dron luego de despegar deba calcular cuál es la siguiente pose que disminuya la incertidumbre del mapa, hasta que el promedio de incertidumbre del mapa sea menor que un valor determinado, de ser así, el dron aterriza.

Dado que el objetivo es crear un algoritmo que busca elegir una pose nueva para mejorar el resultado del SLAM, se usará un algoritmo estándar de SLAM: para la localización se utilizará filtro extendido de Kalman, utilizando *landmarks* artificiales, a definir, y para el mapeo se utilizará un método de mallas geométricas de tres dimensiones, que utilice y actualice su incertidumbre.

Para simplificar el problema se redujo la complejidad del mapa a un escenario virtual simplificado de un cubo, sin obstáculos y su incertidumbre a una variable de confirmación.

3.2. Diseño de los módulos

Para lograr el objetivo general de la investigación se necesita un programa que determine la pose óptima y controle al dron/robot. Para conseguir esto se necesita los siguientes módulos:

Mapa Un módulo que guarde el mapa y permita modificar su incertidumbre.

Optimizador Un módulo que calcule la siguiente pose óptima usando la información del mapa.

Controlador Un módulo que comanda los actuadores del dron para desplazarse a la pose objetivo considerando el margen de error aceptable establecido.

Odometría Un módulo que estime y guarde la pose actual del dron.

Para el robot se necesita que cumpla los siguientes requisitos:

- Flote y se sustente en el aire.
- Un tamaño adecuado para maniobrar en una habitación, menos de un metro cuadrado.
- Sensores que permitan estimar su propia pose.
- Un sensor de medición de distancia frontal.
- Un sensor que permita detectar un punto de referencia absoluto.

Además, se necesita un computador que ejecute el programa y se comunique con el robot

3.3. Arquitectura del sistema

Para lograr que el dron siga el camino óptimo se requieren varios pasos. Al usar el estándar de ROS, se necesita un nodo que calcule la pose óptima usando el mapa y la pose actual, luego un nodo que acepte esta pose y se asegure dentro de cierto rango que el dron llegue a esa pose. Para esto se necesitan dos componentes, un nodo que se encargue de control del dron y un nodo que se encargue de estimar la pose. Para esto se usó la odometría interna y códigos QR. En la figura 3.1 se encuentra el diseño explicado.

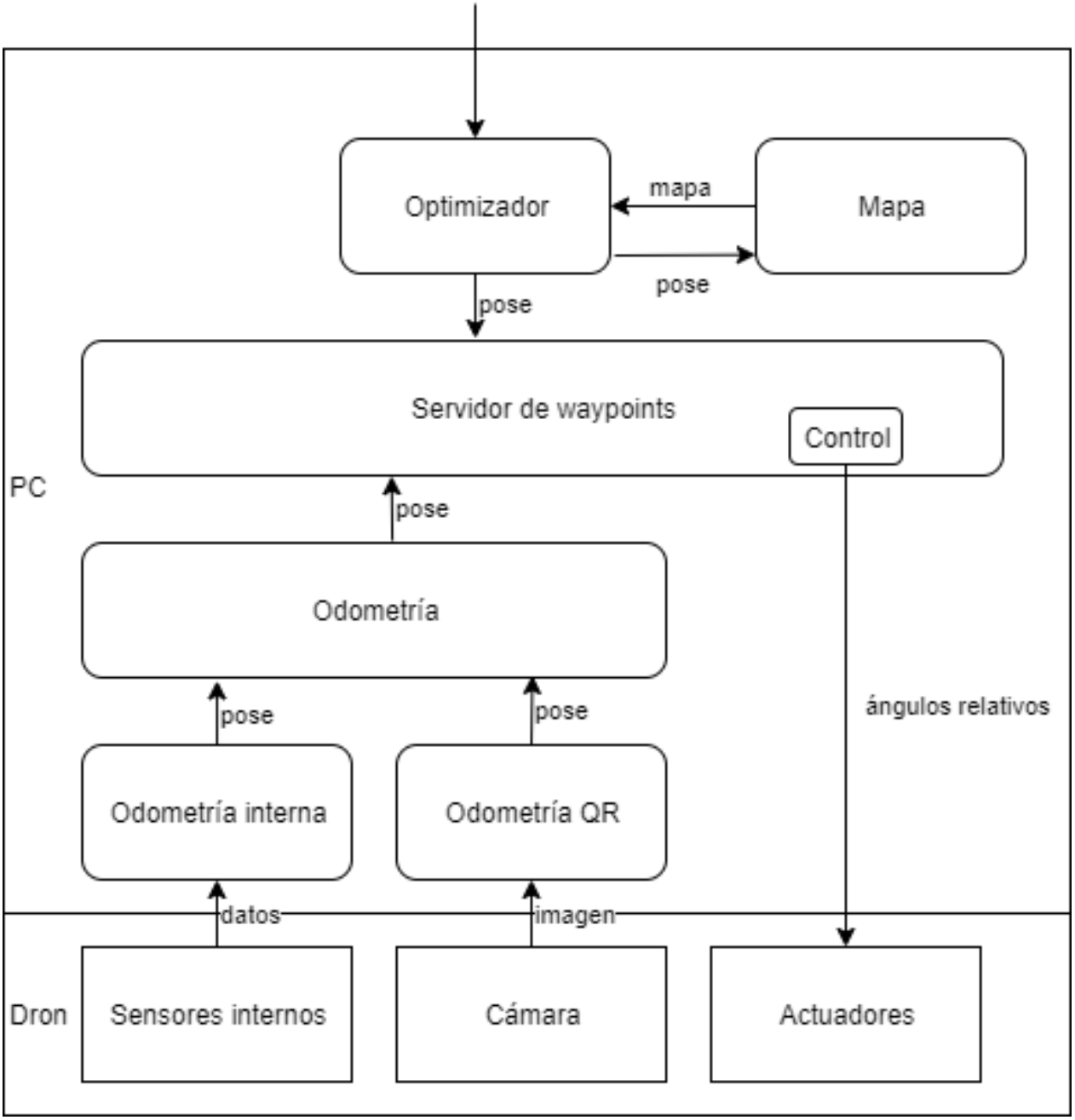


Figura 3.1: UML básico de las componentes

3.4. Odometría

Los módulos de odometría se encargan de acumular los datos de los sensores internos y estimar la pose del dron.

3.4.1. Sensores internos

Para lograr estimar la pose del dron usando los sensores internos (magnetómetro y osciloscopio) se implementa un nodo que envía información de odometría usando el nodo existente en el proyecto de control del AR.Drone con ROS[1], que no se encuentra disponible para el simulador. Se usa la información disponible en el tópico *navdata*. Se cambian los signos y unidades de medida de la información original, se multiplica la diferencia de tiempo entre el mensaje anterior y el actual, contra la velocidad en eje x e y. En el eje z se aprovecha la información obtenida por el sonar inferior, que es muy preciso y no es afectado por la acumulación de error, pero tiene un límite de 3 metros.

Dado que se posee información absoluta de la orientación gracias a un magnetómetro, nos permite saber el ángulo para descomponer cada vector de velocidades, dado que mide fuerzas magnéticas indicando el norte magnético, lo que le permite determinar una orientación absoluta, lo que a su vez nos da un sistema de referencia con respecto al mapa del mundo.

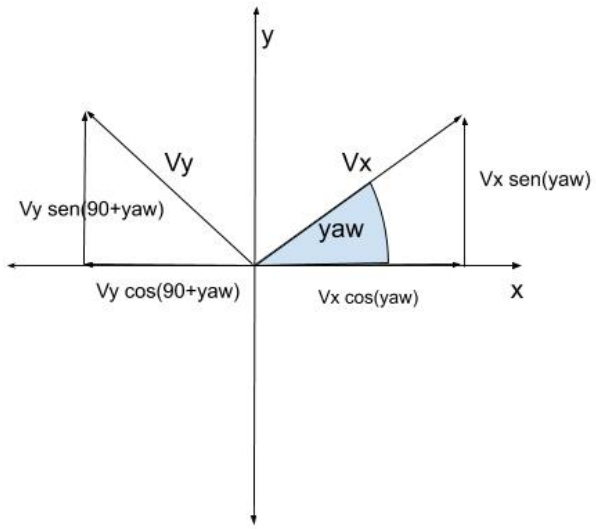


Figura 3.2: Descomposición geométrica de velocidades para el cambio de sistema de referencia

$$Velocidad \times Tiempo = Distancia$$
$$Velocidad_x \times Tiempo = Distancia_x$$

[1] https://github.com/AutonomyLab/ardrone_autonomy/blob/indigo-devel/src/ardrone_driver.cpp#L684

Descomponemos las velocidades en el eje, gracias a la figura 3.2.

$$(\overrightarrow{V_x}\cos(yaw) + \overrightarrow{V_y}\cos(yaw + 90))\Delta t = D_x$$

Dado identidad trigonométrica $\cos(\alpha + 90) = -\sin(\alpha)$.

$$(\overrightarrow{V_x}\cos(yaw) - \overrightarrow{V_y}\sin(yaw))\Delta t = D_x$$

Recordar acumular la distancia $X_{\mathrm{i}-1} + D_x = X_{\mathrm{i}}$.

$$(\overrightarrow{V_x}\cos(yaw) - \overrightarrow{V_y}\sin(yaw))\Delta t + X_{\mathrm{i}-1} = X_{\mathrm{i}}$$

Siguiendo la misma lógica con respecto al eje y.

$$(V_x\sin(yaw) + V_y\sin(yaw + 90))\Delta t = D_y$$

y dado la relación trigonométrica $\sin(yaw + 90) = \cos(yaw)$.

$$(V_x\sin(yaw) + V_y\cos(yaw))\Delta t = D_y$$

Remplazando y recordando acumular la distancia.

$$(V_x\sin(yaw) + V_y\cos(yaw))\Delta t + Y_{\mathrm{i}-1} = Y_{\mathrm{i}}$$

Notar que se debe asegurar que los vectores de velocidad tengan el sentido correcto con respecto al sentido del magnetómetro, o sea, si al estar en 0° con respecto el magnetómetro y se mueve hacia la derecha la velocidad debería ser positiva, lo mismo aplica en el eje y. Luego de pruebas en el simulador el resultado es que los signos están correctos en ambas velocidades.

3.4.2. Odometría visual

Para lograr estimar una pose usando la imagen disponible desde la cámara inferior se detectan: el código QR, la información del código y la posición relativa del código QR en la imagen. Dado que la información del código QR contiene la posición absoluta y con la posición relativa de la imagen se puede calcular la posición absoluta del dron.

El nodo visp_auto_tracker[2] dispone tanto la información sobre el contenido del código QR y la posición relativa con respecto del centro de la imagen. Se maneja la orientación del código QR para que esté alineado con el *groundtrue*, de tal forma que si el dron y el código QR están alineados nos permite usar el ángulo del eje z para descomponer las distancias.

$$Distancia\ percibida + \mathrm{d}istancia\ reportada\ por\ QR = posición\ en\ \mathrm{el}\ \mathrm{d}ron$$

[2]http://wiki.ros.org/visp_auto_tracker

$$D_p + D_{qr} = P$$

Se descompone la distancia relativa gracias al ángulo relativo entre el código QR y el dron.

$$P_x = X_p \cos(\alpha) + Y_p \cos(\alpha + 90)$$

Recordando la relación trigonométrica. $\cos(\alpha + 90) = -\sin(\alpha)$

$$P_x = X_p \cos(\alpha) - Y_p \sin(\alpha)$$

Componiendo tenemos.

$$X_p \cos(\alpha) - Y_p \sin(\alpha) + QR_x = X$$

Para el eje *y* seguimos la misma lógica.

$$P_y + QR_y = Y$$

Descomponemos la distancia reportada por la cámara.

$$X_p \sin(\alpha) + Y_p \sin(\alpha + 90)$$

Recordando la identidad trigonométrica. $\sin(90 + \alpha) = \cos(\alpha)$

$$X_p \sin(\alpha) + Y_p \cos(\alpha)$$

Remplazando tenemos:

$$X_p \sin(\alpha) + Y_p \cos(\alpha) + QR_y = Y$$

3.4.3. Unión de odometría

Para lograr estimar la pose del dron usando la pose obtenida por los nodos anteriores y usando el filtro extendido de Kalman, se unió la información de odometría implementada en la sección 3.4.1, la información de la *IMU* del AR.Drone y la información de la odometría visual implementada en el capítulo 3.4.2, de tal forma que (puesto en las configuraciones) considere solo las posiciones absolutas que vengan del módulo de odometría visual (que es considerada *groundtrue*) y usando la velocidad que proviene de la *IMU*. Para introducir los datos al nodo se configuran los parámetros de partida del nodo indicado.

3.5. Controlador

Para ROS se encuentra disponible una librería para controlar el AR.Drone 2.0 (ardrone_autonomy[3]) donde la función principal envía comandos al tópico cmd_vel que permite mover hacia adelante y hacia atrás en el eje X, la derecha y la izquierda en el eje Y, hacia arriba y hacia abajo en el eje Z, y rotar con respecto al sentido del reloj y contra el sentido del reloj en el eje angular Z.

El nodo de control provee una lista de parámetros para controlar el comportamiento que incluye el ángulo máximo permitido en cada eje, y así se limita la velocidad. Además, están los comandos para aterrizar, despegar y *reset* que son proxy de lo disponible con el AR.Drone. Se usa la librería ardrone_tutorials_gettings_started que posee un nodo en Python para controlar el AR.Drone.

El AR.Drone posee un tópico donde publica información de odometría que no se encuentra disponible en el simulador, se tuvo que implementar porque se utiliza para la localización, que se encuentra como referencia en el repositorio[4].

3.6. Servidor de *waypoints*

El servidor de *waypoints* se encarga de recibir en tiempo real la pose del dron y dar órdenes a los actuadores para lograr que el dron alcance la pose objetivo. Para lograr llegar al objetivo, se diseña el controlador en 2 fases:

- La fase de alto nivel de planeación de vuelo, donde analiza la distancia y la orientación relativa al objetivo con la actual para aprovechar las características del dron durante el vuelo como la cámara frontal.
- La fase de bajo nivel que se le entrega a la distancian relativa a los PID por cada vector de movimiento (x, y, w, z) y se envían los resultados del PID al controlador del dron.

Se diseñó el plan de vuelo del dron en 4 fases implementado como un *switch case*:

- Girar a objetivo: Se reorienta el dron para que apunte hacia el objetivo, logrado por calcular el ángulo entre la diferencia del eje y y el eje x.
  - Se entra en esta fase si la diferencia entra la orientación actual del dron y la orientación que debería tener para avanzar hacia el objetivo es menor que 5 grados.
- Avanzar al objetivo: Se envía el valor *pitch* haciendo que el dron se mueva en sentido del eje X.
  - Si la distancia euclidiana del objetivo y el dron es mayor que 2 metros entonces se entra en esta fase.
- Movimiento en dos ejes de movimiento (x,y): Transforma el objetivo a sistema de referencia del dron y avanza en ambos ejes.
  - Si la distancia euclidiana del objetivo y el dron es menor que 2 metros entonces se entra en esta fase.

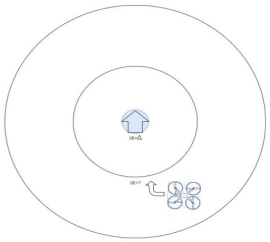
[3] http://wiki.ros.org/ardrone_autonomy

[4] https://github.com/AutonomyLab/ardrone_autonomy/blob/d35607beb211f56d8ce4aafa45115a1be34c97c2/src/ardrone_driver.cp

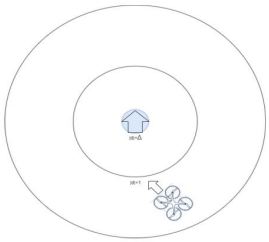
- Reorientación: Una vez que se encuentra cerca del punto y si la velocidad es despreciable (menor a 0,1[m/s]), entonces el dron se reorienta con respecto a la pose objetivo.
  - Si el dron se encuentra en el margen aceptado con respecto el objetivo se entra en esta fase.
- Si se encuentra dentro del margen de error durante toda la reorientación y las velocidades medidas en la odometría son menores al error de medición, se considera que se logró el objetivo.

Durante todo el flujo anterior se corrige la altitud comparando la diferencia en el PID correspondiente.

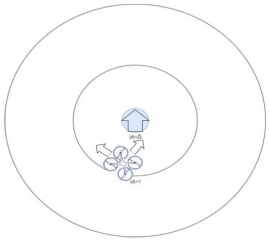
Dado que es muy difícil estar en la posición exacta de la pose objetivo por variadas razones (incertidumbre de la posición, perturbación por el viento), se acepta que el dron esté dentro de un margen de error predefinido, que en este caso es 20 [cm] o 5 grados sexagesimal de error en los ejes de referencia y los ángulos.



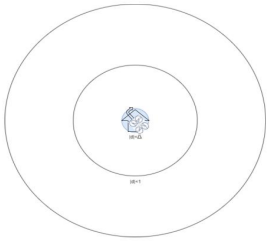
(a) Fase: Girar a objetivo



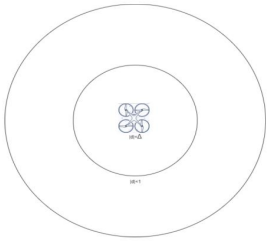
(b) Fase: Avanza al objetivo



(c) Fase: Movimiento en x e y



(d) Fase: Reorientación



(e) Fase: Finalizado

Figura 3.3: Fases de planeación de vuelo

3.7. Optimizador

El nodo optimizador elige según el algoritmo la mejor pose y se encarga de que el dron llegue dentro de un rango de error a esa pose.

Al módulo de nube se le aplica el minimizador que se encuentra disponible en la librería *SciPy*[5], que busca minimizar el valor de la cantidad de puntos aún no vistos.

[5]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize

El minimizador pide una función y un arreglo de números que es un ejemplo de input a la función dada.

```
minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None,
    bounds=None, constraints=(), tol=None, callback=None, options=None )
```

La variable fun es un puntero de una función, que recibe en la primera iteración de entrada $x0$ (y usa el valor como base para iterar) y debe retornar el valor a minimizar. La variable $method$ es un *string* que indica cual método usar de lo que están disponible en la librería *SciPy*.

A *minimize* se le entregan una función que hace una copia de la nube descrita en la sección 3.8, recibe el valor del optimizador y revisa los puntos que vería dado la pose entregada por el optimizador. Luego la función le entrega al optimizador la cantidad de puntos no vistos. Dado que es un minimizador se buscó minimizar la cantidad de puntos no vistos, y el optimizador continúa iterando en este proceso hasta que completa el número máximo de iteraciones definido por el $method$, retornando la variación de $x0$ que logró el menor valor de la fun. Luego se aplica esta variación a $x0$ al objeto de nube original. Posteriormente se itera el proceso anterior con máximo 500 iteraciones o se revisen todo el mapa.

Al *minimize* se le entregan valores (poses) de sugerencia ($x0$). Se crean 2 tipos de poses: las poses cardinales y las poses recomendadas usando información de la nube. Para las poses cardinales se crearon 16 poses, que corresponden a las esquinas interiores a un metro de distancia de la nube y cuatro poses centrales de las "murallas".

Si luego de quinientos intentos fallidos de encontrar una pose, que logre disminuir la cantidad de puntos no vistos, se llama a la función de recomendación de poses 3.8.2 que remplaza a las poses de sugerencia en el optimizador.

Para el optimizador se usó la librería de *SciPy* y principalmente tres métodos: Powell, COBYLA y Nelder-Mead, que son los métodos que lograron revisar todo el mapa. El resto de los métodos probados (CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, trust-constr, dogleg, trust-ncg, trust-exact, trust-krylov) o no lograban revisar todo el mapa o necesitaban el jacobiano[6] de la fun. Dado que el modelo del mundo es discreto, fun es una función escalonada por lo tanto la derivada es una delta de Dirac[7].

3.8. Definición y almacenado del mapa

El nodo mapa se encarga de almacenar y actualizar el mapa. Para generar un estándar de mapa se usa la estructura de nube de puntos $p[] = (x, y, z)$, una variable adicional que denota si el punto fue visto, y un arreglo $normal[] = (x, y, z)$ que representa la normal del punto con el mismo índice.

[6]El jacobiano es la matriz de las derivadas parciales de las variables de la función

[7]Una función que tiende a infinito en un valor y, para cualquier otro valor de x, es igual a 0.

3.8.1. Cono de visión del dron

El sonar emite un pulso sonoro y luego mide el tiempo que se demora en recibir dicho pulso, por lo cual la visión se simplifica como si fuera un cono, para determinar qué puntos de la nube son vistos por el dron, se usa la inecuación de un cono donde parte desde el punto $(0, 0)$ con el eje alineado con el eje x , $((x^2+y^2)*(cos(\theta)^2)-z^2*(sin(\theta)^2))$ con las restricciones $F(x, y, z) < 0\ ,\ 0 < z < h$ donde el cono con altura h y un ángulo de apertura 2θ.

Pero existe el problema que el dron puede estar en cualquier orientación y en cualquier posición en (x, y, z). Para solucionar esto, se realiza el cambio de sistema de referencia, de tal forma que el dron se encuentre en el centro del sistema de referencia, usando los mismos conocimientos que se usa en el módulo de odometría de localización 3.4.1. Para el problema de que la inecuación está alineada al eje x se consideró que no hay una interdependencia entre ejes por lo cual se cambió el x por el z, como se muestra en la inecuación $((z^2 + y^2) * (cos(\theta)^2) - x^2 * (sin(\theta)^2))$ con las restricciones $F(x, y, z) < 0\ ,\ 0 < x < h$.

Un ejemplo visual del cálculo de la inecuación se muestra en figura 3.4, donde el dron es representado por un punto con una flecha y los puntos no vistos son representados por el color rojo y los puntos vistos son representados con color azul y verde.

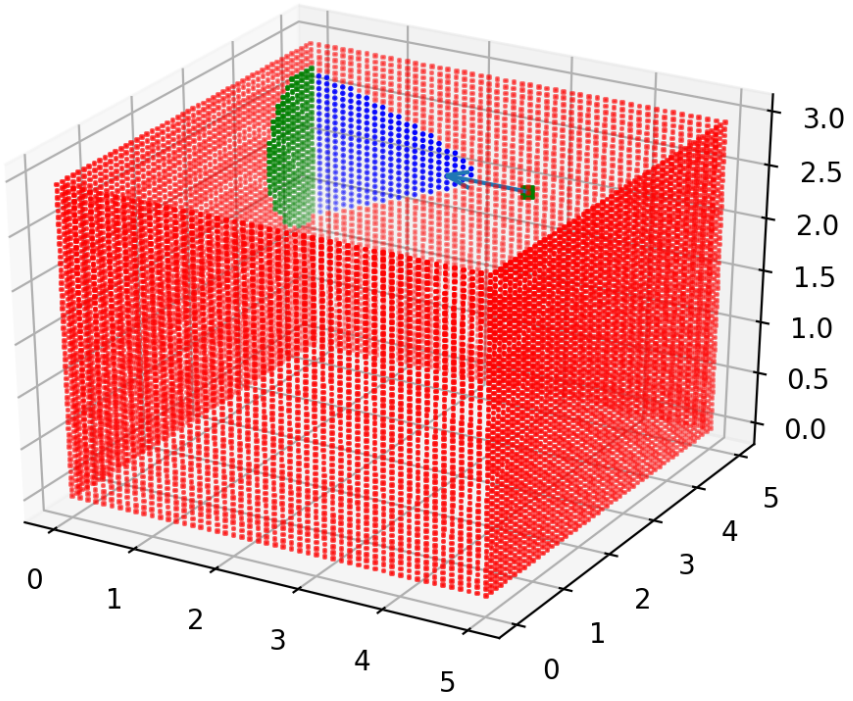


Figura 3.4: Ejemplo de cono de visión del dron, donde los puntos verdes representan los puntos vistos válidos, los puntos azules son puntos dentro de cono de visión, pero con un ángulo mayor a 90 entre la normal y la orientación del dron, y los puntos rojos están contenidos en el cono de visión del dron. Una flecha muestra la orientación del dron y el dron es representado por un cubo verde.

3.8.1.1. Ángulo entre normal y dron

Como otra restricción de simulación se requiere que el dron se encuentre mirando directamente (de frente) al conjunto de puntos para que sean considerados como vistos, para esto se usa la normal de los puntos y la orientación del dron usando la formula $\theta = arccos(A \cdot B/|A| * |B|)$ donde A es la normal del punto y B es la orientación del dron, en ambos vectores se puede ignorar el eje Z dado que las murallas son paralelas al eje z.

3.8.2. Recomendación de pose por la nube

Se necesita agregar una función que entregue una pose que permita al dron revisar nuevos puntos, dado que en los últimos pasos de optimización la probabilidad de encontrar un punto sin revisar

es muy baja. Para cada punto sin revisar se cuentan los vecinos sin revisar por búsqueda por anchura con una distancia máxima de 10, eligiendo el primer punto con más puntos de vecinos que tiene sin revisar.

Capítulo 4

Implementación

En este capítulo se presenta la implementación de los módulos de: odometría, control, optimización y mapa. Cumpliendo las funcionalidades descritas en el capítulo 3. Notar que se usará las definiciones de ROS descritos en sección 2.2.1.

4.1. Odometría

En esta sección se describe la implementación de los tres módulos encargados de la odometría: odometría interna, odometría usando códigos QR y unión de odometría, estos módulos se encargan de recibir diferentes mediciones del dron y generar la pose actual en el sistema de referencia absoluta.

4.1.1. Odometría por velocidades

Se implementa un nodo llamado *simulador_odometry* que recibe información odométrica (velocidades y orientación en los ejes desde la *IMU*) desde el dron y calcula la pose actual.

A continuación, se explica el código donde se encuentran las fórmulas, se incluye las transformaciones dado que los grados están en radianes y las velocidades en cm/s como es el estándar en ROS.

La variable *navdata* proviene del tópico suscrito */ardrone/navdata* que origina del nodo *gazebo* descrito en el capítulo 3.4.1 que contiene las velocidades, descompuestas en los ejes de referencia del dron y la orientación estimada del dron. Tomando las velocidades y la orientación del dron se estima la pose del dron en el sistema de referencia del mundo. Publicando la pose estimada del dron por el tópico */ardrone/odometry*.

```
1     vx=navdata.vx / 1000.0
2     vy=navdata.vy / 1000.0
3     yaw=(navdata.rotZ / 180.0) * pi
```

```
    if self.last_receive_time:
      delta_t = navdata.header.stamp.to_sec() - self.last_receive_time.to_sec()
      self.odometry[0] += (cos(yaw) * vx - sin(yaw) * vy) * delta_t
      self.odometry[1] += (sin(yaw) * vx + cos(yaw) * vy) * delta_t
    self.last_receive_time = navdata.header.stamp
```

4.1.2. Odometría con códigos de rápida respuesta

El nodo *qr_marker* estima la pose del dron usando la posición del código QR en la imagen, obtenida del tópico */visp_auto_tracker/object_position* y la información contenida en el código QR disponible en el tópico */visp_auto_tracker/code_message* y publica la pose estimada en tópico */landmark_odom*.

Dado que el dron puede estar observando el código QR desde diferentes posiciones y orientaciones relativas al código QR, es necesario descomponer la posición del código QR en la imagen, por lo cual se calcula la posición del dron usando el código QR, se toma la posición del código QR en la imagen y se suma a la posición del código QR contenido en él (contenida en *qr_code*).

```
      odom_point=Point(
              x=(cos(rad_yaw)*pos.x - sin(rad_yaw)*pos.y )+self.qr_code.x,
              y=(sin(rad_yaw)*pos.x + cos(rad_yaw)*pos.y )+self.qr_code.y,
              z=self.qr_code.z + pos.z
              )
```

4.1.3. Unión de odometría

Robot_localization[1] es una librería disponible para ROS que acepta una cantidad arbitraria de mediciones de odometría y las unifica usando un filtro extendido de Kalman para crear una estimación de la pose.

Para utilizar la librería se necesita modificar un archivo de configuración, donde se definen la cantidad y los nombres de los tópicos a integrar y qué partes del tópico se utilizan.

```
<!--  The order of the values is: x, y, z,
                                  roll, pitch, yaw,
                                  vx, vy, vz,
                                  vroll, vpitch, vyaw,
                                  ax, ay, az. -->

    <param name="odom0" value="/landmark_odom"/>
    <rosparam param="odom0_config">[true, true, true,
                                    false,  false, false,
                                    false, false, false,
                                    false, false, false,
                                    false, false, false]
    </rosparam>
```

[1] http://wiki.ros.org/robot_localization

```
14
15      <param name="odom1" value="/ardrone/odometry"/>
16      <rosparam param="odom1_config">[false, false, false,
17                                      true, true, true,
18                                      true, true, true,
19                                      false, false, false,
20                                      false, false, false]
21      </rosparam>
22
23      <param name="imu0" value="/ardrone/imu"/>
24      <rosparam param="imu0_config">[ false, false, false,
25                                      true, true, true,
26                                      false, false, false,
27                                      true, true, true,
28                                      false, false, false]
29      </rosparam>
30
31      <rosparam param="initial_state">[0.0,  0.0,  0.0,
32                                       0.0,  0.0,  0.0,
33                                       0.0,  0.0,  0.0,
34                                       0.0,  0.0,  0.0,
35                                       0.0,  0.0,  0.0]
36      </rosparam>
37
38      <param name="frequency" value="50"/>
39
40      <param name="map_frame" value="world"/>
41      <param name="odom_frame" value="odom"/>
42      <param name="base_link_frame" value="ardrone_base_link"/>
```

Para declarar una odometría se incluye el parámetro *odomX* o *imuX* donde *X* se remplaza por un número identificador. Otro parámetro importante es *[odomX_config]* o *[imuX_config]* (reemplazando X el número identificador) que determina qué componentes de la odometría se usa en el filtro en el orden mostrado a continuación, reemplazando los valores con *true* o *false*, siendo *v* velocidad y *a* aceleración, considerando que una pose es posición y orientación.

```
1 <!--  The order of the values is: x, y, z,
2                                    roll, pitch, yaw,
3                                    vx, vy, vz,
4                                    vroll, vpitch, vyaw,
5                                    ax, ay, az. -->
```

En *odom0* se pone la odometría visual, es decir solo la información sobre su posición que son las tres distancias al eje de referencia.

```
1       <param name="odom0" value="/landmark_odom"/>
2       <rosparam param="odom0_config">[true, true, true,
3                                       false,  false, false,
4                                       false, false, false,
5                                       false, false, false,
6                                       false, false, false]
7       </rosparam>
```

En *odom1* se define la odometría de sensores internos que sólo considera las velocidades del AR.Drone junto con la orientación del AR.Drone.

```
    <param name="odom1" value="/ardrone/odometry"/>
    <rosparam param="odom1_config">[false, false, false,
                                    true, true, true,
                                    true, true, true,
                                    false, false, false,
                                    false, false, false]
    </rosparam>
```

La *IMU* configurada en el valor *imu0* unifica la orientación y la velocidad angular.

```
    <param name="imu0" value="/ardrone/imu"/>
    <rosparam param="imu0_config">[ false, false, false,
                                    true, true, true,
                                    false, false, false,
                                    true, true, true,
                                    false, false, false]
    </rosparam>
```

A través del parámetro *initial_state* se establece el supuesto que el AR.Drone parte en el centro del eje de referencia.

```
    <rosparam param="initial_state">[0.0,  0.0,  0.0,
                                     0.0,  0.0,  0.0,
                                     0.0,  0.0,  0.0,
                                     0.0,  0.0,  0.0,
                                     0.0,  0.0,  0.0]
    </rosparam>
```

Se usa una frecuencia mayor que la frecuencia de las odometrías (30 [Hz]). Se intentó poner una frecuencia igual a la de las odometrías, pero resultaba con mensaje de error por no poder utilizar los datos, por lo cual se fija la frecuencia en 50 [Hz].

```
    <param name="frequency" value="50"/>
```

Se definen los nombres de los sistemas de referencia. La única diferencia es con *base_link_frame*, que es el *frame* que sigue al AR.Drone a *ardrone_base_link* para ser consistente con el controlador que lo define de esa manera.

```
    <param name="map_frame" value="world"/>
    <param name="odom_frame" value="odom"/>
    <param name="base_link_frame" value="ardrone_base_link"/>
```

4.2. Controlador del dron

Como segundo componente en la ejecución del camino, se usa la odometría obtenida en la sección anterior 4.1.3 para lograr llegar a un punto objetivo. Dado que sólo se puede controlar los ángulos relativos del dron al plano horizontal, se complica el objetivo de llegar hasta el punto que se busca.

Puesto que no existe una fórmula directa entre el ángulo del dron y la distancia recorrida, se usará un PID para controlar el ángulo sobre las variables controlables *roll*, *pitch*, velocidad en *yaw*,

velocidad en el eje z, como se ve en la figura 4.1.

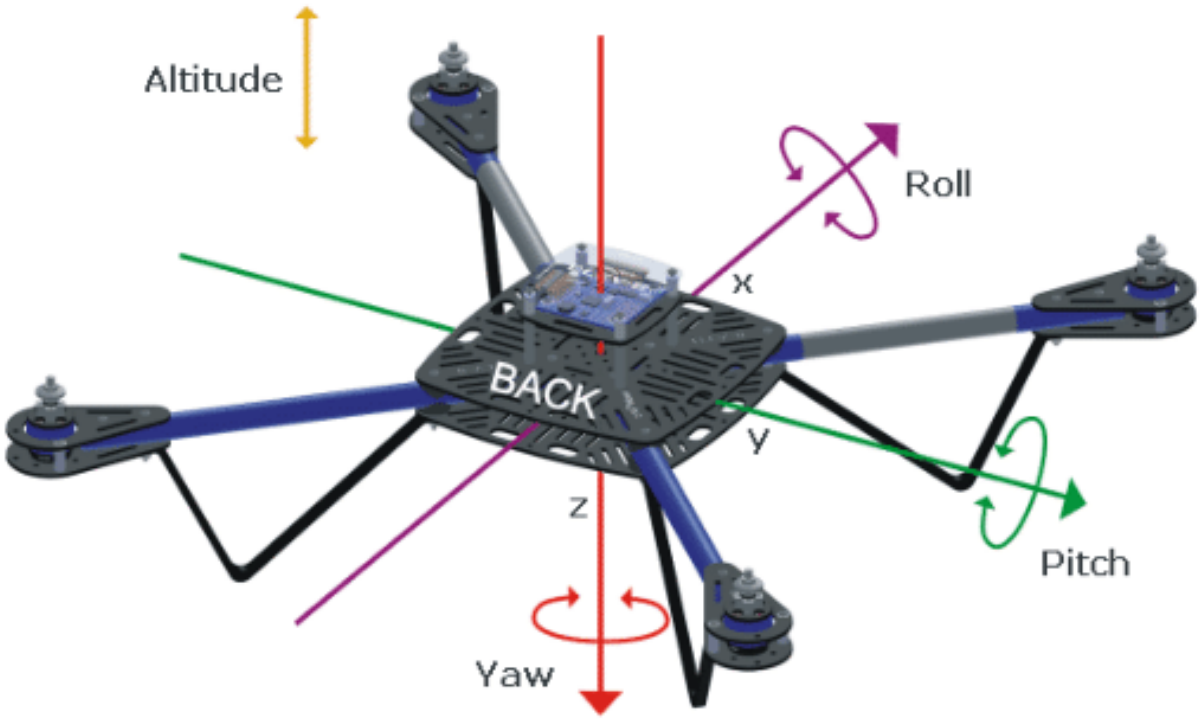


Figura 4.1: Representación de los ángulos de navegación en un dron y su correspondiente eje [2].

4.2.1. Controlador Proporcional, Integral y Derivativo

Para lograr que el dron llegue a la posición objetivo se controlan los ángulos con respecto al plano horizontal del dron, que en consecuencia cambia la velocidad del dron y dado que no se puede conocer todas las condiciones que afectan al dron, como la velocidad y dirección del viento, se usa un controlador PID para controlar los ángulos del drone.

4.2.2. Ajuste de parámetros del Controlador Proporcional, Integral y Derivativo

Para el ángulo *yaw*:

- La variable medida es el ángulo con respecto el eje z dado gracias a la odometría (magnetómetro y QR).
- El punto deseado es el ángulo objetivo dado el *input* del controlador.
- El error es la diferencia entre el ángulo medido por la odometría y el objetivo dado por el controlador.

- La variable controlada es el valor *yaw* enviado al proxy de ROS del controlador del dron.

Para el ángulo *roll* y *pitch*:

- La variable medida es la posición en el eje y o x respectivamente dada por la odometría.
- El punto deseado es la componente y o x de la posición objetivo dado el *input* del controlador.
- El error es la diferencia entre la odometría y el objetivo de las componentes y o x.
- La variable controlada es el valor *roll* y *pitch* enviado al proxy de ROS del controlador del dron.

Para la altura z:

- La variable medida será la posición en el eje z (medida en odometría).
- El punto deseado es la componente z de la posición objetivo dado el *input* del controlador.
- El error es la diferencia entre la odometría y el objetivo del componente z.
- La variable controlada es el valor *linear.z* enviado al proxy de ROS del controlador del dron.

4.2.3. Determinación de constantes y experimentación

La heurística que se utiliza para determinar las constantes es establecer las constantes PID a cero. A continuación, se incrementa el valor P hasta que la cantidad de ciclos encuentre un mínimo. Luego incrementar I hasta que el valor de salida encuentre un mínimo. Finalmente incrementar D si es que se necesita.

Para esto se realizan experimentos probando con valores correspondientes a la exponencial desde menos cinco a cuatro, dado que los valores menores que uno generan mayores cambios y por otro lado los valores grandes no afectan. Por esto se utiliza la función exponencial, por su densidad en el tramo cero excluyente a uno inclusivo (]0,1]) y su crecimiento en mayor a uno es exponencial lo cual permite demostrar la inexistente diferencia entre números mayores de 10.

Son cuatro PID que se implementan sobre:

- *Roll* que tiene relación al eje y (en el sistema de referencia del dron).
- *Pitch* que tiene relación con el eje de referencia de x (en el sistema de referencia del dron).
- Velocidad sobre *yaw* que tiene relación con la orientación del dron.
- Velocidad del eje z.

Para los experimentos sobre cada PID se usan 10 repeticiones. A estas constantes se le calculan el promedio y la desviación estándar de la cantidad de ciclos que se demora en lograr el objetivo, además, se consideró una distancia media de 3 metros de distancia.

Para la constante integral y derivada se prueban 5 incrementos más (15 en total) que la constante proporcional, dado que no se tenía experiencia sobre números grandes.

4.2.3.1. Eje x

Para el eje x, que se relaciona con el ángulo *pitch*, se le dio de objetivo al dron un punto a 3 metros de distancia en el eje x.

Constante proporcional Se elige de la Tabla 4.1 el valor 20 para la constante proporcional dado ser el menor valor que logra el menor tiempo y número de iteraciones considerando la desviación estándar, se descarta el valor 403 porque tiene mayor desviación estándar en el número de iteraciones que el valor 20.

	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
$0,051$	1542	16	51,4	0,5
0,083	999	7	33,2	0,2
0,168	485	1	16,15	0,05
0,401	190,4	0,8	6,31	0,03
1,033	77,1	1,3	2,54	0,04
2,751	61,6	0,7	2,02	0,02
7,422	52,7	0,6	1,72	0,02
20,118	47,4	1,0	1,55	0,03
54,631	49,1	1,1	1,60	0,04
148,446	48,0	2,1	1,57	0,07
403,462	47,1	1,5	1,54	0,05

Tabla 4.1: Tabla con datos que se usa para determinar la constante proporcional del eje x producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante integral Se elige de la tabla 4.2 el valor 0,13 para la constante integral dado que es el primer valor en lograr 48,2 segundos aproximado con menor desviación estándar que 0.04, notar que dado la poca diferencia entre mediciones se elijó 48,2 para evitar números mayores que generen oscilaciones, pero aun si no cambia el resultado.

X_I	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,0183	72,4	3,5	2,3	0,1
0,0497	48,2	3,8	1,5	0,1
0,135	48,2	1,4	1,58	0,05
0,367	47,9	1,6	1,56	0,05
1,000	47,5	1,8	1,55	0,06
2,7182	48,0	1,6	1,56	0,05
7,389	47,6	1,6	1,55	0,05
20,085	47,4	1,4	1,54	0,04
54,598	48,0	2,8	1,56	0,09
148,413	48,1	1,7	1,57	0,05
403,428	47,7	1,4	1,55	0,04
1096,633	48,0	1,5	1,56	0,05
2980,957	47,9	1,4	1,56	0,04
8103,083	47,5	1,5	1,55	0,04
22026,465	48,0	1,1	1,56	0,03

Tabla 4.2: Tabla con datos que se usa para la determinar la constante integral del eje x producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante Derivada Se eligió de la tabla 4.3 el valor 0.367 para la constante diferencial dado ser el menor valor que logra el menor tiempo y número de iteraciones, se elije 0.367 para evitar números mayores que generen oscilaciones, pero aun si no cambia el resultado, además no se elige 8103,083 dado que tiene una desviación estándar de 5.5 que es mayor al valor elegido.

X_D Constante	Tiempo [s] Promedio	σ	Iteraciones Promedio	σ
0,018	72,8	4,1	2,3	0,1
0,0497	47,8	1,7	1,56	0,05
0,135	48,3	1,5	1,57	0,05
0,367	47,2	1,0	1,54	0,03
1,0	47,9	1,5	1,56	0,05
2,718	48,1	1,8	1,57	0,06
7,389	49,9	2,6	1,63	0,08
20,085	48,8	1,8	1,59	0,06
54,598	53,1	13,6	1,7	0,4
148,413	49,9	7,3	1,6	0,2
403,428	52,5	15,8	1,7	0,5
1096,633	50,9	7,1	1,6	0,2
2980,957	48,3	1,7	1,57	0,05
8103,083	46,8	5,5	1,5	0,1
22026,465	48,4	1,2	1,58	0,04

Tabla 4.3: Tabla con datos que se usa para la determinar la constante derivada del eje x producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

4.2.3.2. Eje y

Para el eje y, que se relaciona con el ángulo *roll*, se le dio de objetivo al dron 3 metros de distancia en el eje y para determinar experimentalmente las constantes.

Constante Proporcional Se elige de la tabla 4.4 el valor 2,718 para la constante proporcional dado ser el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar, notar que no se elige 1 a pesar de que tiene menor promedio, tiene peor desviación estándar que 2,718.

Y_P	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,0183	653,8	698,4	21,7	23,3
0,0497	58,5	19,9	1,9	0,6
0,1353	48,3	1,7	1,57	0,05
0,367	47,5	1,8	1,55	0,06
1,000	47,3	1,4	1,54	0,04
2,718	47,4	0,9	1,54	0,03
7,389	56,6	12,7	1,8	0,4
20,085	3070,7	8976,4	102,3	299,3
54,598	3147,5	8952,1	104,9	298,5
148,413	3355,1	8897,7	111,8	296,8

Tabla 4.4: Tabla con datos que se usan para la determinar la constante proporcional del eje y producidos experimentalmente. Se presentan los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante Integral Se elige de la tabla 4.5 el valor 7,389 para la constante integral dado ser el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar.

Y_I	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,018	71,4	14,3	2,3	0,4
0,049	51,7	9,8	1,6	0,3
0,135	49,0	2,9	1,60	0,09
0,367	48,3	1,6	1,57	0,05
1,000	51,8	10,1	1,6	0,3
2,718	50,0	7,5	1,6	0,2
7,389	47,6	1,5	1,55	0,05
20,085	80,5	16,3	2,6	0,5
54,598	93,4	23,2	3,0	0,7
148,413	83,6	27,1	2,7	0,9
403,428	77,1	22,8	2,5	0,7
1096,633	69,4	22,0	2,2	0,7
2980,957	79,6	21,0	2,6	0,7
8103,083	91,2	34,6	3,0	1,1
22026,465	73,3	23,9	2,4	0,7

Tabla 4.5: Tabla con datos que se usan para determinar la constante integral del eje y producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante diferencial Se elige de la tabla 4.6 el valor 0,048 para la constante diferencial dado que es el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar.

Y_D	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,018	79,1	8,6	2,6	0,2
0,049	56,8	10,9	1,8	0,3
0,135	61,9	27,4	2,0	0,9
0,367	58,8	15,6	1,9	0,5
1,000	59,2	23,2	1,9	0,7
2,718	129,5	160,2	4,2	5,3
7,389	131,7	172,9	4,3	5,7
20,085	133,5	139,8	4,4	4,6
54,598	285,7	303,3	9,4	10,1
148,413	250,9	274,5	8,3	9,1
403,428	253,5	311,3	8,4	10,3
1096,633	479,1	753,5	15,9	25,1
2980,957987	662,2	766,5	22,0	25,5
8103,083928	123,2	129,5	4,0	4,3
22026,46579	89,1	49,5	2,9	1,6

Tabla 4.6: Tabla con datos que se usa para determinar de la constante diferencial del eje y producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

4.2.3.3. Eje z

Para el eje z, que se relaciona con la velocidad de altura, se le da de objetivo al dron 3 metros de distancia en el eje z para determinar experimentalmente las constantes.

Z_P	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,051	1524,9	13,9	50,9	0,3
0,083	990,5	2,4	32,99	0,08
0,168	484,3	4,7	16,1	0,1
0,401	246,7	11,5	8,2	0,1
1,033	195,4	1,3	6,48	0,04
2,751	187,2	1,3	6,21	0,04
7,422	181,2	1,9	6,01	0,06
20,118	183,1	1,5	6,07	0,05
54,631	179,8	4,1	6,01	0,04
148,446	181,3	1,8	6,01	0,06
403,462	182,3	2,0	6,05	0,06

Tabla 4.7: Tabla con datos que se usa para determinar la constante proporcional del eje z producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante Proporcional Se elige de la tabla 4.7 el valor 7,4 para el valor de la constante proporcional dado que es el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar.

Constante Integral Se elige de la tabla 4.8 el valor 1,000 para la constante integral dado que es el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar, se evitan los números mayor a 100 que pueden generar oscilaciones, notar que todos los valores de constante entregan tiempos y numero de iteraciones iguales dentro de la desviación estándar por lo cual todos son válidos.

Z_I	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,018	187,1	2,5	6,21	0,08
0,049	181,8	1,7	6,03	0,05
0,135	182,0	1,6	6,03	0,05
0,367	181,3	3,1	6,03	0,07
1,000	181,7	1,3	6,02	0,04
2,718	181,6	1,4	6,02	0,04
7,389	182,7	1,6	6,06	0,05
20,085	181,9	1,6	6,03	0,05
54,598	181,7	1,5	6,03	0,05
148,413	181,5	1,5	6,02	0,05
403,428	181,3	2,2	6,03	0,06
1096,633	181,6	1,4	6,02	0,04
2980,957	181,8	1,7	6,03	0,05
8103,083	181,8	1,5	6,03	0,05
22026,465	181,8	1,6	6,03	0,05

Tabla 4.8: Tabla con datos que se usan para determinar la constante integral del eje z producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante Derivada Se decide de la tabla 4.9 que la constante diferencial fuera 0 dado que ningún valor genera una mejora en comparación con los resultados obtenidos para la determinación de la constante integral (recordando que cuando se realizó el experimento para la constante diferencial el valor de la constante integral es 0), se evitaron los números mayor a 100 que pueden generar oscilaciones.

Z_D	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,018	189,0	4,1	6,2	0,1
0,049	197,7	46,4	6,5	1,5
0,135	193,9	39,1	6,4	1,2
0,367	201,0	60,0	6,6	2,0
1,000	186,0	12,0	6,1	0,4
2,718	190,0	27,0	6,3	0,9
7,389	186,9	9,5	6,1	0,3
20,085	190,7	8,8	6,3	0,2
54,598	188,3	10,8	6,2	0,3
148,413	180,8	2,1	5,99	0,07
403,428	180,8	3,5	6,02	0,05
1096,633	182,0	1,2	6,03	0,04
2980,957	181,3	0,4	6,01	0,01
8103,083	181,3	1,4	6,02	0,04
22026,465	182,0	1,2	6,03	0,04

Tabla 4.9: Tabla con datos que se usa para determinar la constante derivada del eje z producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

4.2.3.4. Eje w

Para el eje w, que se relaciona con el ángulo *yaw*, se le da de objetivo al dron que girara 90 grados para determinar experimentalmente las constantes.

Constante Proporcional Se elige de la tabla 4.10 el valor 7,422 para la constante proporcional por ser el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar.

W_P	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,051	1941,3	45,7	64,7	1,4
0,083	1229,6	6,9	40,9	0,2
0,168	593,0	15,9	19,7	0,5
0,401	249,8	2,9	8,30	0,09
1,033	95,9	3,9	3,1	0,1
2,751	42,1	0,9	1,37	0,03
7,422	34,4	1,0	1,11	0,03
20,118	36,1	1,4	1,17	0,04
54,631	37,6	1,4	1,22	0,04
148,446	37,0	1,1	1,20	0,03
403,462	36,8	1,1	1,19	0,03

Tabla 4.10: Tabla con datos que se usan para determinar la constante proporcional de la variable *yaw* producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante Integral Se elige de la tabla 4.11 el valor 0,0183 para la constante integral dado que es el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar.

W_I	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,0183	36,1	0,9	1,2	0,2
0,0497	36,3	0,9	1,18	0,03
0,135	36,6	1,1	1,19	0,04
0,367	36,3	1,4	1,18	0,04
1,000	37,1	1,5	1,21	0,05
2,718	36,6	1,1	1,19	0,03
7,389	37,9	1,2	1,23	0,03
20,085	37,5	1,1	1,22	0,03
54,598	37,6	0,9	1,22	0,02
148,413	37,0	1,0	1,20	0,03
403,428	37,4	0,9	1,21	0,03
1096,633	38,3	0,7	1,25	0,02
2980,957	38,0	2,0	1,24	0,06
8103,083	36,7	2,3	1,19	0,08
22026,465	37,3	1,1	1,21	0,03

Tabla 4.11: Tabla con datos que se usa para determinar la constante integral de la variable *yaw* producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

Constante Diferencial Se elige de la tabla 4.12 el valor 0,018 para la constante diferencial dado que es el menor valor que logra el menor tiempo y número de iteraciones con menor desviación estándar.

W_D	Tiempo [s]		Iteraciones	
Constante	Promedio	σ	Promedio	σ
0,018	35,2	0,6	1,14	0,02
0,049	37,2	0,6	1,20	0,02
0,135	35,8	0,8	1,16	0,02
0,367	38,8	2,4	1,26	0,08
1,000	35,6	1,4	1,15	0,04
2,718	38,5	4,2	1,28	0,08
7,389	40,2	3,0	1,3	0,1
20,085	37,8	1,1	1,22	0,03
54,598	38,1	2,6	1,23	0,08
148,413	39,3	7,4	1,2	0,2
403,428	37,4	1,3	1,21	0,04
1096,633	37,2	0,9	1,20	0,03
2980,957	36,6	0,6	1,18	0,02
8103,083	37,2	0,9	1,20	0,03
22026,465	36,9	1,7	1,19	0,05

Tabla 4.12: Tabla con datos que se usa para determinar la constante diferencial de la variable *yaw* producidos experimentalmente. Se presenta los valores promedio y desviación estándar (σ) del tiempo en segundos y la cantidad de ciclos que toma en llegar a la pose objetivo. La fila destacada con gris es el valor elegido como constante.

4.2.3.5. Resumen de constantes de los ejes

En la tabla 4.13 se muestra las constantes elegidas en cada eje.

Ejes Constantes	X	Y	Z	W
Proporcional	20,118	0,367	7,422	7,422
Integral	0,135	7,389	1,000	0,0183
Derivada	0,367	0,049	0,018	0,018

Tabla 4.13: Valores de las constantes del los PIDs de cada eje escogidas empíricamente

4.2.4. Control del dron y determinación del arribo de él a la pose objetivo

El nodo llamado *server_waypoint* recibe una pose objetivo o una lista de poses objetivo por el servicio con nombre *server_waypoint*. Primero transforma la pose objetivo al sistema de referencia del dron (de esta forma es clara la relación con las variables controlables) y se calcula la diferencia entre la pose objetivo y la pose actual. Luego esta diferencia se envía al controlador PID correspondiente al eje y el resultado del PID se envía al controlador del dron.

```
1     def set_objective_response(self, msg):
2         r = rospy.Rate(self._rate)
3         rospy.logdebug('{}: goal received'.format(self._action_name))
4         self._goal = msg
5 
6         i = 0
7         j = 0
8         self.calculate_diff()
9         while not self.is_arrived_goal() and i < self._limit_loop:
10            self.next_goal()
11            i = i + 1
12            # put all distances in 0
13            self.calculate_diff()
14            rospy.logdebug('{}: set_objective_response w1 - {}'.format(self.
    _action_name, i < self._limit_loop))
15            while not self.is_arrived() and j < self._limit_loop:
16                rospy.logdebug('{}: set_objective_response w2 - {}'.format(
    self._action_name, j < self._limit_loop))
17                self.calculate_diff()
18                rospy.logdebug(self._diff)
19                j = j + 1
20                # check that preempt has not been requested by the client
21                if self._as.is_preempt_requested():
22                    rospy.logdebug('{}: Preempted'.format(self._action_name))
23                    self._as.set_preempted()
24                    i = self._limit_loop
25                    break
26                self.control_path()
27                self.feedback()
28                r.sleep()
29                self._controller.SetCommand(roll=0, pitch=0, yaw_velocity=0,
    z_velocity=0)
30                rospy.sleep(1)
31                rospy.logdebug('{}: trajectory goal archived'.format(self.
    _action_name))
32        rospy.loginfo('{}: goal archived in {} operantions'.format(self.
    _action_name, j))
33        self._controller.SetCommand(roll=0, pitch=0, yaw_velocity=0,
    z_velocity=0)
34        _result = action_controller.msg.MultiDofFollowJointTrajectoryResult()
35        self._as.set_succeeded(_result)
```

En la línea 4 se guarda la pose objetivo entregada a la función en la variable del objeto *_goal*. Luego se calcula la diferencia entre la posición actual y el objetivo en *calculate_diff*. Entra en un *loop* mientras no se encuentre en la última pose objetivo o se cumpla la cantidad máxima de iteraciones. De esta forma se permite un camino con múltiples poses.

```
    def set_objective_response(self, msg):
        r = rospy.Rate(self._rate)
        rospy.logdebug('{}: goal received'.format(self._action_name))
        self._goal = msg

        i = 0
        j = 0
        self.calculate_diff()
        while not self.is_arrived_goal() and i < self._limit_loop:
```

Se calcula si se encuentra cerca de la pose; de ser así se toma la siguiente pose como objetivo.

```
            self.next_goal()
            i = i + 1
```

Se recalcula la distancia entre el objetivo y la pose actual.

```
            # put all distances in 0
            self.calculate_diff()
```

Se revisa si se ha logrado la pose objetivo o si se cumple la cantidad máxima de iteraciones.

```
            rospy.logdebug('{}: set_objective_response w1 - {}'.format(self._action_name, i < self._limit_loop))
            while not self.is_arrived() and j < self._limit_loop:
```

Se recalcula la distancia entre el objetivo y la pose actual.

```
                rospy.logdebug('{}: set_objective_response w2 - {}'.format(self._action_name, j < self._limit_loop))
                self.calculate_diff()
                rospy.logdebug(self._diff)
                j = j + 1
```

Se revisa si el objetivo no ha sido cancelado.

```
                # check that preempt has not been requested by the client
                if self._as.is_preempt_requested():
                    rospy.logdebug('{}: Preempted'.format(self._action_name))
                    self._as.set_preempted()
                    i = self._limit_loop
                    break
```

El nodo *server_waypoint* recibe una pose y se llama a *control_path* que calcula y envía la orden de movimiento al dron.

```
                self.control_path()
```

Se calcula la retroalimentación para que esté disponible de ser requerido y cumplir con lo requerido por el diseño de ROS. Si se logra llegar a un objetivo intermedio o final se envía una señal al dron para que se detenga (línea 3) y se espera 1 segundo para asegurarse que el dron se detenga. En caso de que no haya podido detenerse y haya continuado de largo, esta situación será detectada en la siguiente iteración del *loop* y se corregirá.

```
                self.feedback()
                r.sleep()
                self._controller.SetCommand(roll=0, pitch=0, yaw_velocity=0,
    z_velocity=0)
                rospy.sleep(1)
                rospy.logdebug('{}: trajectory goal archived'.format(self.
    _action_name))
        rospy.loginfo('{}: goal archived in {} operantions'.format(self.
    _action_name, j))
```

Finalmente se envía la orden de éxito del objetivo como parte del protocolo de servicios de ROS.

```
        self._controller.SetCommand(roll=0, pitch=0, yaw_velocity=0,
    z_velocity=0)
        _result = action_controller.msg.MultiDofFollowJointTrajectoryResult()
        self._as.set_succeeded(_result)
```

4.2.4.1. Metodología de acercamiento al objetivo

A continuación, se analiza el código de la función *control_path* que envía las instrucciones al dron, esta función es llamada por *set_objective_response* en la línea 26 en el capítulo 4.2.4.

```
    def control_path(self):
        # Set the args to Set command thus ensuring that previous command dont
     affects actual command
        self._ac = {"roll": 0, "pitch": 0, "yaw": 0, "z": 0}
        # If not in the correct x,y point the pitch
        if self.is_advance_reach():
            # self.close()
            # if the dron is near (2 meters) of the objective
            if self.is_closer_x_y_final_section():
                if not self._is_in_closer_state:
                    self._pid_y.reset()
                    self._pid_x.reset()
                    self._is_in_closer_state = True
                self.close()
            else:
                if self._is_in_closer_state:
                    self._is_in_closer_state = False
                self.point_advance()
        # if not in the correct self._objective orientation then rotate
        elif not self.is_angle_correct():
            rospy.logdebug('{}: correction orietation'.format(self.
    _action_name))
            if fabs(self._odom["vx"])+fabs(self._odom["vy"]) < 0.05:
                self.calcule_yaw(self._objective["w"])
        else:
            rospy.logdebug('{}: correct position, check ending'.format(self.
    _action_name))
        # If not in the correct z the change altitude
        if not self.is_z_correct():
            rospy.logdebug('{}: change elevation'.format(self._action_name))
            # vel = self._vel_max * min(1.0, fabs(self._diff["z"]))
            # self._ac["z"] = copysign(vel, self._diff["z"])
```

```
30              # Actualize PID controller
31              vel = self._pid["z"].update_PID(self._diff["z"])
32              self._ac["z"] = copysign(vel, self._diff["z"])
33          if True:
34              self._controller.SetCommand(roll=self._ac["roll"],
35                                          pitch=self._ac["pitch"],
36                                          yaw_velocity=self._ac["yaw"],
37                                          z_velocity=self._ac["z"])
```

Se asignan los valores a 0 para que los valores anteriores no afecten.

```
1   def control_path(self):
2       # Set the args to Set command thus ensuring that previous command dont
    affects actual command
3       self._ac = {"roll": 0, "pitch": 0, "yaw": 0, "z": 0}
```

Se determina si el dron está orientado al objetivo.

```
1       # If not in the correct x,y point the pitch
2       if self.is_advance_reach():
3           # self.close()
4           # if the dron is near (2 meters) of the objective
```

Se determina si el dron está a 2 metros de la pose objetivo, de ser así se revisa que la variable *_is_in_closer_state* sea verdadera y luego se llama la función *close* que guarda los valores para que el dron avance tanto en el eje x como el eje y hacia el objetivo. De no estar a 2 metros del objetivo se revisa el *flag* que esté marcado y se calculan y guardan los valores, con la función *point_advance* línea 11, al dron para que avance en el eje x.

```
1           if self.is_closer_x_y_final_section():
2               if not self._is_in_closer_state:
3                   self._pid_y.reset()
4                   self._pid_x.reset()
5                   self._is_in_closer_state = True
6               self.close()
7           else:
8               if self._is_in_closer_state:
9                   self._is_in_closer_state = False
10              self.point_advance()
```

Se determina si el eje x del sistema de referencia del dron está alineado con el objetivo, aceptando un pequeño margen de error. De no ser así y si el dron se encuentra detenido, se calcula la orientación que el dron tiene que tener para alinearse con el punto objetivo y se guarda el valor. Si el dron no cumplió ninguna condición quiere decir que logró llegar al objetivo.

```
1       # if not in the correct self._objective orientation then rotate
2       elif not self.is_angle_correct():
3           rospy.logdebug('{}: correction orietation'.format(self.
    _action_name))
4           if fabs(self._odom["vx"])+fabs(self._odom["vy"]) < 0.05:
5               self.calcule_yaw(self._objective["w"])
6       else:
7           rospy.logdebug('{}: correct position, check ending'.format(self.
    _action_name))
```

Además, se calcula si el dron se encuentra a la altura objetiva, de no ser así se determina la velocidad necesaria gracias el PID en la línea 7, guardando la velocidad y asignándole el signo de la diferencia entre la pose objetivo y la pose actual.

```
1         # If not in the correct z the change altitude
2         if not self.is_z_correct():
3             rospy.logdebug('{}: change elevation '.format(self._action_name))
4             # vel = self._vel_max * min(1.0, fabs(self._diff["z"]))
5             # self._ac["z"] = copysign(vel, self._diff["z"])
6             # Actualize PID controller
7             vel = self._pid["z"].update_PID(self._diff["z"])
8             self._ac["z"] = copysign(vel, self._diff["z"])
```

Finalmente se llama a la función *SetComand* del controlador del dron, las variables de la función son los valores calculados por los PID en cada eje durante la ejecución de *control_path*.

```
1         if True:
2             self._controller.SetCommand(roll=self._ac["roll"],
3                                         pitch=self._ac["pitch"],
4                                         yaw_velocity=self._ac["yaw"],
5                                         z_velocity=self._ac["z"])
```

4.3. Ciclo de cálculo de pose óptima y actualización de la incertidumbre

En el ciclo principal, dada una lista de métodos de optimización a probar, se crea una lista para guardar el valor de la métrica de optimización en cada ciclo por cada método y se entra en un ciclo de quinientos (500) iteraciones donde el método de optimización elige la pose del dron que disminuya la incertidumbre del mapa.

```
1     list_of_method = [ "Powell", "COBYLA", "Nelder-Mead"]
2     list_of_accuracy = dict()
3     for method in list_of_method:
4         try:
5             log_file = open("/home/enzo/ros/catkin_ws/src/dron_path/log/
    optimization_{}_rad.txt".format(method), "w")
6             po = PoseOptimization(log_file)
7             pa = PointsAcomulator()
8             x0 = [3.0, 0.5, 0.2, 0.0]
9             position = {"x": 3.5, "y": 0.5, "z": 0.5}
10            orientation = {"x": 2.5, "y": 0.5, "z": 0.0}
11            drone_path = list()
12            bounds = [(0, 5), (0, 5), (0, 3), (0, 2 * pi)]
13            list_of_cardinal_point = list()
14            x_range = [2, 3]
15            y_range = [2, 3]
16            z_range = [1, 2]
17            angles = [0, pi/2, pi, 3*pi/2]
18
19            for z in z_range:
20                point = [x_range[0], y_range[0], z, angles[3]]
```

```
                list_of_cardinal_point.append(point)
                point = [x_range[0], y_range[0], z, angles[2]]
                list_of_cardinal_point.append(point)

            for z in z_range:
                point = [x_range[1], y_range[0], z, angles[3]]
                list_of_cardinal_point.append(point)
                point = [x_range[1], y_range[0], z, angles[0]]
                list_of_cardinal_point.append(point)

            for z in z_range:
                point = [x_range[0], y_range[1], z, angles[1]]
                list_of_cardinal_point.append(point)
                point = [x_range[0], y_range[1], z, angles[2]]
                list_of_cardinal_point.append(point)

            for z in z_range:
                point = [x_range[1], y_range[1], z, angles[1]]
                list_of_cardinal_point.append(point)
                point = [x_range[1], y_range[1], z, angles[0]]
                list_of_cardinal_point.append(point)


            x0 = [3.0, 2.5, 1.5, 0.0]
            list_of_cardinal_point.append(x0)
            x1 = [2.5, 3.0, 1.5, pi * 0.5]
            list_of_cardinal_point.append(x1)
            x2 = [2.0, 2.5, 1.5, pi * 1.0]
            list_of_cardinal_point.append(x2)
            x3 = [2.5, 2.0, 1.5, pi * 1.5]
            list_of_cardinal_point.append(x3)
            counter_cardinal_point = 1

            x = list_of_cardinal_point[0]

            maxiter = 4 * 1000
            options = {"maxiter": maxiter}
            last = 0
            count_of_cicles_without_optimization = 0
            margin = 0.0001
            list_of_accuracy[method] = list()

            for z in range(500):
                res = minimize(po.optimize_function, x, bounds=bounds, method=
method, options=options)
```

Primero se define la lista de métodos usados para optimizar que se intentaran.

```
        list_of_method = [ "Powell", "COBYLA", "Nelder-Mead"]
        list_of_accuracy = dict()
```

Se itera por cada método en *list_of_method*.

```
    for method in list_of_method:
            try:
```

Se crean los objetos que maneja la optimización

```
            log_file = open("/home/enzo/ros/catkin_ws/src/dron_path/log/
    optimization_{}_rad.txt".format(method), "w")
            po = PoseOptimization(log_file)
            pa = PointsAcomulator()
```

Se crean una pose donde el dron donde los métodos puedan partir.

```
            position = {"x": 3.5, "y": 0.5, "z": 0.5}
            orientation = {"x": 2.5, "y": 0.5, "z": 0.0}
```

Se crea una lista para guardar las poses óptimas.

```
            drone_path = list()
```

Se definen los límites de los cuales el método puede probar poses.

```
            bounds = [(0, 5), (0, 5), (0, 3), (0, 2 * pi)]
```

Se crea una lista donde se guardan las poses que se recomiendan a los métodos.

```
            list_of_cardinal_point = list()
```

Se definen los componentes de poses recomendados.

```
            x_range = [2, 3]
            y_range = [2, 3]
            z_range = [1, 2]
            angles = [0, pi/2, pi, 3*pi/2]
```

Se definen las poses recomendadas y se agregan a la lista *list_of_cardinal_points*.

```
            for z in z_range:
                point = [x_range[0], y_range[0], z, angles[3]]
                list_of_cardinal_point.append(point)
                point = [x_range[0], y_range[0], z, angles[2]]
                list_of_cardinal_point.append(point)

            for z in z_range:
                point = [x_range[1], y_range[0], z, angles[3]]
                list_of_cardinal_point.append(point)
                point = [x_range[1], y_range[0], z, angles[0]]
                list_of_cardinal_point.append(point)

            for z in z_range:
                point = [x_range[0], y_range[1], z, angles[1]]
                list_of_cardinal_point.append(point)
                point = [x_range[0], y_range[1], z, angles[2]]
                list_of_cardinal_point.append(point)

            for z in z_range:
                point = [x_range[1], y_range[1], z, angles[1]]
                list_of_cardinal_point.append(point)
                point = [x_range[1], y_range[1], z, angles[0]]
                list_of_cardinal_point.append(point)
```

```
24
25
26              x0 = [3.0, 2.5, 1.5, 0.0]
27              list_of_cardinal_point.append(x0)
28              x1 = [2.5, 3.0, 1.5, pi * 0.5]
29              list_of_cardinal_point.append(x1)
30              x2 = [2.0, 2.5, 1.5, pi * 1.0]
31              list_of_cardinal_point.append(x2)
32              x3 = [2.5, 2.0, 1.5, pi * 1.5]
33              list_of_cardinal_point.append(x3)
34              counter_cardinal_point = 1
```

Se hace *pop* de la primera pose recomendada.

```
1               x = list_of_cardinal_point[0]
```

Se define la cantidad máxima de iteraciones que un método puede intentar por cada pose recomendada.

```
1               maxiter = 4 * 1000
2               options = {"maxiter": maxiter}
```

Se definen variables para revisar cuantas iteraciones el método realiza sin lograr mejoras en la disminución de incertidumbre.

```
1               last = 0
2               count_of_cicles_without_optimization = 0
```

Se crea en el diccionario una lista que contiene los valores estadísticos sobre la cantidad de ciclos para lograr 25 %, 50 %, 75 %, 100 % de certidumbre.

```
1               list_of_accuracy[method] = list()
```

Se itera 500 veces esto para asegurar que el método termine.

```
1               for z in range(500):
```

Se llama a la función *minimize* que se encarga de llamar al método de optimización y revisa cuanto disminuye la incertidumbre cada iteración.

```
1                   res = minimize(po.optimize_function, x, bounds=bounds, method=
    method, options=options)
```

4.4. Mapa

En esta sección se describen como se construye la nube de puntos y las funciones que calculan los puntos vistos por el sonar y la que recomienda una pose que detecte la mayor cantidad de puntos no visto.

4.4.1. Constructor del objeto mapa

En el constructor de la clase se definen variables auxiliares para la construcción y estandarización de la nube de puntos

```
    def __init__(self):
        self.t = (2*51+2*49)*50
        self.p = np.zeros((self.t, 3))
        self.n = np.zeros((self.t, 3))
        self.pb = np.zeros(self.t, dtype=bool)
        self.previousPointCheck = np.zeros(self.t, dtype=bool)
        self.pointLastCheck=list()
        self.theta = 15.0
        self.h = 3.0
        self.n_points_face=0
        self.triangulation=None
        self.counter_index_ref=None
        self.simulate_cloud()
```

La variable *t* guarda la numero de puntos totales de la nube de puntos.

```
    def __init__(self):
        self.t = (2*51+2*49)*50
```

Se crean variables p, n y pb que contienen un arreglo de tres *int* que representan: punto de la nube de puntos, la normal estimada en el punto y al booleano correspondiente a la condición si el punto fue visto, respectivamente.

```
        self.p = np.zeros((self.t, 3))
        self.n = np.zeros((self.t, 3))
        self.pb = np.zeros(self.t, dtype=bool)
```

Se definen 2 variables para guardar información de ciclo anteriores para depuración y para graficar.

```
        self.previousPointCheck = np.zeros(self.t, dtype=bool)
        self.pointLastCheck=list()
```

Se definen variables para simular el sonar, *theta* es el ángulo interno del cono, *h* es la altura de cono.

```
        self.theta = 15.0
        self.h = 3.0
```

Se definen las variables *n_points_face* que contiene la cantidad de puntos en una muralla, *counter_index_ref* que guarda una referencia entre índices de puntos y *triangulation* que guarda el objeto resultado de la triangulación, las tres variables son usadas para calcular la vecindad de puntos en las intersecciones.

```
        self.n_points_face=0
        self.triangulation=None
        self.counter_index_ref=None
```

Al finalizar la función *simulate_cloud* se llama que usa las variables para construir la nube de puntos descrito en la sección posterior 4.4.2

```
        self.simulate_cloud()
```

4.4.2. Definición y construcción de la geometría del espacio

En el código mostrado a continuación se crea una nube de puntos con forma cuadrada como se define en la sección 3.8; la nube creada se guarda en *self* del objeto *PointsAcomulator* a la cual esta función pertenece.

```
    def simulate_cloud(self):
        #3__________2
        #|          |
        #|          |
        #|0__________|1
        #4
        cont = 0
        max_x = 5
        max_y = 5
        max_z = 3
        n_seg = 50.0
        for x in range(0, int(n_seg + 1)):
            for z in range(0, int(n_seg)):
                if x == 0:
                    self.n[cont] = [0.5, 0.5, 0]
                elif x == n_seg:
                    self.n[cont] = [-0.5, 0.5, 0]
                else:
                    self.n[cont] = [0, 1, 0]
                xt = x * (max_x / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [xt, 0.0, zt]
                cont = cont + 1
        for y in range(1, int(n_seg)):
            for z in range(0, int(n_seg)):
                yt = y * (max_y / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [5.0, yt, zt]
                self.n[cont] = [-1, 0, 0]
                cont = cont + 1
        for x in range(int(n_seg), -1, -1):
            for z in range(0, int(n_seg)):
                if x == 0:
                    self.n[cont] = [0.5, -0.5, 0]
                elif x == n_seg:
                    self.n[cont] = [-0.5, -0.5, 0]
                else:
                    self.n[cont] = [0, -1, 0]
                xt = x * (max_x / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [xt, 5.0, zt]
                cont = cont + 1
        for y in range(int(n_seg - 1), 0, -1):
```

```
            for z in range(0, int(n_seg)):
                yt = y * (max_y / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [0.0, yt, zt]
                self.n[cont] = [1, 0, 0]
                cont = cont + 1
        from scipy.spatial import Delaunay

        points = self.p[:2550, 0:3:2]
        self.triangulation = Delaunay(points)
        self.counter_index_ref = dict()
        for i in range(int(n_seg)):
            self.counter_index_ref[i] = i + 2500
            self.counter_index_ref[i + 2500] = i
```

Al inicio de la función *simulate_cloud* se declaran variables temporales para manejar y estandarizar el tamaño del mapa, siendo una habitación de cinco metros de ancho *max_x*, cinco metros de largo *max_y* y una altura de tres metros *max_z*. Además de un contador *cont* que se usa para confirmar la cantidad de puntos en la nube. Se incluye la variable *n_seg* que define la cantidad de filas y columnas en cada muralla.

```
    def simulate_cloud(self):
        #3__________2
        #|          |
        #|          |
        #|0_________|1
        #4
        cont = 0
        max_x = 5
        max_y = 5
        max_z = 3
        n_seg = 50.0
```

Se generan los puntos pertenecientes a la primera muralla en el eje *x*, se itera, por la densidad *n_seg*, en el eje *y* y en el eje *z*. En dichas iteraciones se le asigna una normal al punto, usando la normal como si fuera una muralla. Después se calcula la posición del punto, manteniendo fijo el valor *y=0*, usando el número de iteración en cada eje multiplicado la altura máxima dividido la densidad de las murallas. Luego se guarda en la variable del objeto *p*.

```
        for x in range(0, int(n_seg + 1)):
            for z in range(0, int(n_seg)):
                if x == 0:
                    self.n[cont] = [0.5, 0.5, 0]
                elif x == n_seg:
                    self.n[cont] = [-0.5, 0.5, 0]
                else:
                    self.n[cont] = [0, 1, 0]
                xt = x * (max_x / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [xt, 0.0, zt]
                cont = cont + 1
```

Terminando la primera muralla se definen los puntos de la segunda muralla siguiendo el eje *y*, usando la misma lógica anterior, se itera, n_seg veces, en el eje *x* y el eje *z*. En dichas iteraciones se

le asigna una normal, como si fuera parte de una muralla. Después se calcula la posición del punto, manteniendo fijo el valor $x = 5$, usando el número de iteración en cada eje multiplicado la altura máxima dividido la densidad de las murallas. Luego se guarda en la variable del objeto p.

```
        for y in range(1, int(n_seg)):
            for z in range(0, int(n_seg)):
                yt = y * (max_y / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [5.0, yt, zt]
                self.n[cont] = [-1, 0, 0]
                cont = cont + 1
```

Terminando la segunda muralla se definen los puntos de la tercera muralla siguiendo el eje x, usando la misma lógica anterior, se itera, n_seg veces, en el eje y y en el eje z. En dichas iteraciones se le asigna una normal, como si fuera parte de una muralla. Después se calcula la posición del punto, manteniendo fijo el valor $y = 5.0$, usando el número de iteración en cada eje multiplicado la altura máxima dividido la densidad de las murallas. Luego se guarda en la variable del objeto p.

```
        for x in range(int(n_seg), -1, -1):
            for z in range(0, int(n_seg)):
                if x == 0:
                    self.n[cont] = [0.5, -0.5, 0]
                elif x == n_seg:
                    self.n[cont] = [-0.5, -0.5, 0]
                else:
                    self.n[cont] = [0, -1, 0]
                xt = x * (max_x / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [xt, 5.0, zt]
                cont = cont + 1
```

Terminando la tercera muralla se definen los puntos de la cuarta muralla siguiendo el eje x, usando la misma lógica anterior, se itera, n_seg veces, en los ejes y y z. En dichas iteraciones se le asigna una normal, como si fuera parte de una muralla. Después se calcula la posición del punto, manteniendo fijo el valor $x = 0$, usando el número de iteración en cada eje multiplicado la altura máxima dividido la densidad de las murallas. Luego se guarda en la variable del objeto p.

```
        for y in range(int(n_seg - 1), 0, -1):
            for z in range(0, int(n_seg)):
                yt = y * (max_y / n_seg)
                zt = z * (max_z / n_seg)
                self.p[cont] = [0.0, yt, zt]
                self.n[cont] = [1, 0, 0]
                cont = cont + 1
```

Finalmente se importa desde *SciPy* la función *Delaunay* que retorna un arreglo que contiene un diccionario donde la llave es el índice del punto y los valores son los índices de los puntos que son parte de un triángulo. Además, se crea un diccionario que correlaciona los índices de un extremo de una muralla con los puntos que se encuentran adyacentes en la muralla contigua.

```
        from scipy.spatial import Delaunay

        points = self.p[:2550, 0:3:2]
        self.triangulation = Delaunay(points)
```

```
        self.counter_index_ref = dict()
        for i in range(int(n_seg)):
            self.counter_index_ref[i] = i + 2500
            self.counter_index_ref[i + 2500] = i
```

4.4.3. Exploración del espacio que recorre el dron

Para simular la visión del sonar frontal como visto en la sección 3.8.1 del dron se implementa el siguiente algoritmo.

4.4.3.1. Detección de puntos dentro de cono de visión del dron

La función recibe la pose del dron y el mapa, usando la inecuación del cono se recorre cada punto calculando si cada uno pertenece al cono, se marca cada punto que ha sido visto.

```
        for i in range(0, self.t):
            d_x_o = self.p[i, 0] - position["x"]
            d_y_o = self.p[i, 1] - position["y"]
            d_z = self.p[i, 2] - position["z"]

            cosa = cos(-1.0 * orientation["z"] * pi / 180)
            sina = sin(-1.0 * orientation["z"] * pi / 180)
            #print("cosa ", cosa, " sina " ,sina)

            d_x = d_x_o * cosa - d_y_o * sina
            d_y = d_x_o * sina + d_y_o * cosa

            # 2 calculate the f(x, y, z) = (z ^ 2 + y ^ 2)(cos(theta) ^ 2 - x ^ 2(sin(theta)) ^ 2
            # should complime f(x, y, z) <= 0, h > x > 0,

            cosf = cos(self.theta * pi / 180.0)
            sinf = sin(self.theta * pi / 180.0)

            f = (pow(d_z, 2) + pow(d_y, 2)) * pow(cosf, 2) - pow(d_x, 2) * pow(sinf, 2);
            d_x = d_x_o * cosa - d_y_o * sina
            d_y = d_x_o * sina + d_y_o * cosa
            # 2 calculate the f(x, y, z) = (z ^ 2 + y ^ 2)(cos(theta) ^ 2 - x ^ 2(sin(theta)) ^ 2
            # should comply f(x, y, z) <= 0, h > x > 0,

            cosf = cos(self.theta * pi / 180.0)
            sinf = sin(self.theta * pi / 180.0)
            f = (pow(d_z, 2) + pow(d_y, 2)) * pow(cosf, 2) - pow(d_x, 2) * pow(sinf, 2);
```

Para determinar si un punto en la nube de puntos se encuentra dentro del cono de visión, primero se cambia el sistema de referencia de los datos obtenidos desde odometría, que están en el sistema de referencia del mapa, al sistema de referencia del dron.

```
        for i in range(0, self.t):
            d_x_o = self.p[i, 0] - position["x"]
            d_y_o = self.p[i, 1] - position["y"]
            d_z = self.p[i, 2] - position["z"]

            cosa = cos(-1.0 * orientation["z"] * pi / 180)
            sina = sin(-1.0 * orientation["z"] * pi / 180)
            #print("cosa ", cosa, " sina " ,sina)

            d_x = d_x_o * cosa - d_y_o * sina
            d_y = d_x_o * sina + d_y_o * cosa

            # 2 calculate the f(x, y, z) = (z ^ 2 + y ^ 2)(cos(theta) ^ 2 - x
    ^ 2(sin(theta)) ^ 2
            # should complime f(x, y, z) <= 0, h > x > 0,

            cosf = cos(self.theta * pi / 180.0)
            sinf = sin(self.theta * pi / 180.0)

            f = (pow(d_z, 2) + pow(d_y, 2)) * pow(cosf, 2) - pow(d_x, 2) * pow
    (sinf, 2);
```

Luego se calculan variables temporales de la fórmula explicada en la sección 2.1.

```
            d_x = d_x_o * cosa - d_y_o * sina
            d_y = d_x_o * sina + d_y_o * cosa
            # 2 calculate the f(x, y, z) = (z ^ 2 + y ^ 2)(cos(theta) ^ 2 - x
    ^ 2(sin(theta)) ^ 2
            # should comply f(x, y, z) <= 0, h > x > 0,

            cosf = cos(self.theta * pi / 180.0)
            sinf = sin(self.theta * pi / 180.0)
```

Finalmente se calcula la inecuación que permite determinar si el punto revisado es contenido en el cono de visión.

```
            f = (pow(d_z, 2) + pow(d_y, 2)) * pow(cosf, 2) - pow(d_x, 2) * pow
    (sinf, 2);
```

4.4.3.2. Exclusión de puntos por restricción del sonar

Dada una restricción que tienen los sonares donde no logran detectar un objeto si el ángulo entre la dirección de propagación de la onda y la normal del objeto es mayor a 45 grados, dado que la onda reflejada no llega con suficiente intensidad al detector del dron. Para esto se implementa un algoritmo que excluye los puntos si ángulo entre la normal y la orientación del dron es mayor a 45 grados.

```
            # 3 calculate angle between normal and cone
            # arcos = ((normal) dot product (orientation))/(Euclidean distance
    (normal) * Euclidean distance(orientation))
            # because of the change of reference orinetation is (1,0,0)
            n_x = self.n[i, 0] * cosa - self.n[i, 1] * sina
            n_y = self.n[i, 0] * sina + self.n[i, 1] * cosa
```

```
            n_z = self.n[i, 2]

            dividend = n_x * 1
            divisor = sqrt(pow(n_x, 2)+pow(n_y, 2)+pow(n_z, 2)) * 1
            if divisor == 0:
                print(dividend, " ", divisor)
            p = (acos(dividend / divisor)) * 180.0 / pi

            if f <= 0 and 0 < d_x and d_x < self.h:

                if abs(p-180) < 45:
                    if not self.pb[i]:
                        self.pointLastCheck.append(self.p[i])
                    self.pb[i] = True
            dividend = n_x * 1
            divisor = sqrt(pow(n_x, 2)+pow(n_y, 2)+pow(n_z, 2)) * 1
            if divisor == 0:
                print(dividend, " ", divisor)
            p = (acos(dividend / divisor)) * 180.0 / pi
            if f <= 0 and 0 < d_x and d_x < self.h:
                if abs(p-180) < 45:
                    if not self.pb[i]:
                        self.pointLastCheck.append(self.p[i])
                    self.pb[i] = True
```

Primero se cambia el sistema de referencia de la normal del punto para que sea compatible con la inecuación explicada en la sección 2.1 en las líneas 3,4,5.

```
            # 3 calculate angle between normal and cone
            # arcos = ((normal) dot product (orientation))/(Euclidean distance
    (normal) * Euclidean distance(orientation))
            # because of the change of reference orinetation is (1,0,0)
            n_x = self.n[i, 0] * cosa - self.n[i, 1] * sina
            n_y = self.n[i, 0] * sina + self.n[i, 1] * cosa
            n_z = self.n[i, 2]

            dividend = n_x * 1
            divisor = sqrt(pow(n_x, 2)+pow(n_y, 2)+pow(n_z, 2)) * 1
            if divisor == 0:
                print(dividend, " ", divisor)
            p = (acos(dividend / divisor)) * 180.0 / pi

            if f <= 0 and 0 < d_x and d_x < self.h:

                if abs(p-180) < 45:
                    if not self.pb[i]:
                        self.pointLastCheck.append(self.p[i])
                    self.pb[i] = True
```

Luego se calcula el producto punto entre la normal y la orientación del dron $(1.0.0)$. Dado que el cambio de sistema de referencia está alineado con la orientación del dron, se multiplica la componente *x* de la normal con la orientación.

```
            dividend = n_x * 1
```

Se calcula la distancia euclidiana del vector normal y se multiplica por la distancia euclidiana de la orientación que por definición es 1.

```
            divisor = sqrt(pow(n_x, 2)+pow(n_y, 2)+pow(n_z, 2)) * 1
```

Se confirma que sean valores válidos.

```
            if divisor == 0:
                print(dividend, " ", divisor)
```

Se calcula el ángulo usando el arco-tangente del valor anteriormente calculado y se transforma de radianes a grados por conveniencia.

```
            p = (acos(dividend / divisor)) * 180.0 / pi
```

Finalmente se determina si el valor absoluto del ángulo medido está entre 45 grados, de ser así el punto se considera visto.

```
            if f <= 0 and 0 < d_x and d_x < self.h:
                if abs(p-180) < 45:
                    if not self.pb[i]:
                        self.pointLastCheck.append(self.p[i])
                    self.pb[i] = True
```

4.4.4. Pose recomendada que contiene la mayor cantidad de puntos sin revisar en el mundo

El ciclo principal en la sección 4.3 llama a la función *recommend_pose* cuando no hay un cambio significativo de la cantidad de puntos visto de la nube, ésta función accede el mapa y revisa cada punto contando cuántos puntos vecinos no se han visto, eligiendo el punto con mayor cantidad de vecinos sin ver, retornando la pose del dron a un metro de distancia del punto elegido, usando la normal estimada del punto.

```
        log_neightbor = open("/home/enzo/ros/catkin_ws/src/dron_path/log/
    log_neightbor", "w")

        high_index = None
        high_counter = -1
        record_of_points = list()

        for index in range(self.t):
            if not self.pb[index]:
                # print(index)
                visited_matrix = self.pb.copy()
                points_to_check = deque()
                record_of_points_dirty = list()
                counter = 0
                first_pair = (index, 0)
                points_to_check.append(first_pair)
                record_of_points_dirty.append(first_pair[0])
                while not len(points_to_check) == 0:
                    pair = points_to_check.popleft()
```

```
                    list_of_neighbor = self.request_neighbor(pair[0])
                    # log_neightbor.write("point:{}      neightbor{} \r\n".
format(pair[0], list_of_neighbor))

                    for point in list_of_neighbor:
                        if (not visited_matrix[point]) and pair[1] < 10:
                            points_to_check.append((point, pair[1] + 1))
                            counter = counter + 1
                            record_of_points_dirty.append(point)
                            visited_matrix[point] = True
                if high_counter < counter:
                    high_counter = counter
                    high_index = index
                    record_of_points = record_of_points_dirty.copy()
                    log_neightbor.write("elected point:{}, counter{} \r\n".
format(index, counter))
                else:
                    log_neightbor.write("point:{}, counter{} \r\n".format(
index, counter))

        # return counter
        log_neightbor.close()

        if not high_index == None:
            angle = atan2(-1.0*self.n[high_index,1], -1.0*self.n[high_index
,0])
            return [self.p[high_index, 0] + self.n[high_index,0], self.p[
high_index, 1] + self.n[high_index,1] , self.p[high_index, 2], angle]
        return 0
```

Primero se define un archivo para guardar información de depuración y también variables temporales.

```
        log_neightbor = open("/home/enzo/ros/catkin_ws/src/dron_path/log/
log_neightbor", "w")

        high_index = None
        high_counter = -1
        record_of_points = list()
```

Se itera por todos los puntos en el mapa.

```
        for index in range(self.t):
```

Se revisa si el punto ha sido visto por el dron.

```
            if not self.pb[index]:
```

Se copia la matriz de puntos visitados para marcar puntos vistos en esté ciclo.

```
                visited_matrix = self.pb.copy()
```

Se crea una cola doblemente terminada para registrar los puntos a revisar.

```
                points_to_check = deque()
```

Se crea una dupla guardando el número identificador del punto y su profundidad. Se encola el primer punto en la cola.

```
                first_pair = (index, 0)
                points_to_check.append(first_pair)
```

Mientras la cola no esté vacía se mantiene en el ciclo.

```
                while not len(points_to_check) == 0:
```

Se desencola desde la derecha para seguir el algoritmo de búsqueda por anchura un punto de la cola.

```
                    pair = points_to_check.popleft()
```

Se llama a *request_neighbor* que retorna una lista con los puntos vecinos gracias a la triangulación previamente hecha.

```
                    list_of_neighbor = self.request_neighbor(pair[0])
                    # log_neightbor.write("point:{}      neightbor{} \r\n".
    format(pair[0], list_of_neighbor))
```

Se recorre cada vecino del punto.

```
                    for point in list_of_neighbor:
```

Se asegura que no se haya visto antes esté punto y que el punto no esté a más de 10 puntos de distancia.

```
                        if (not visited_matrix[point]) and pair[1] < 10:
```

Se agrega a la cola la dupla del punto y la distancia del punto de cual es vecino más uno.

```
                            points_to_check.append((point, pair[1] + 1))
```

Se marca el punto como visitado en el arreglo *visited_matrix*.

```
                            visited_matrix[point] = True
```

Se revisa si el punto que se encontró tiene más puntos sin ver que el anteriormente encontrado, de ser así se guarda el índice.

```
                if high_counter < counter:
                    high_counter = counter
                    high_index = index
                    record_of_points = record_of_points_dirty.copy()
                    log_neightbor.write("elected point:{}, counter{} \r\n".
    format(index, counter))
                else:
                    log_neightbor.write("point:{}, counter{} \r\n".format(
    index, counter))
```

Finalmente se calcula la pose del dron que esté a 1 metro de distancia usando la normal del punto, orientado hacia el punto y se retorna esta pose.

```
        if not high_index == None:
            angle = atan2(-1.0*self.n[high_index,1], -1.0*self.n[high_index
    ,0])
            return [self.p[high_index, 0] + self.n[high_index,0], self.p[
    high_index, 1] + self.n[high_index,1] , self.p[high_index, 2], angle]
        return 0
```

Capítulo 5

Mediciones y análisis

En este capítulo se presentan las características del computador utilizado para hacer los experimentos, las formas de los mapas usados en las pruebas y los resultados de éstos.

5.1. Ambiente de pruebas

Las características del computador utilizado para las pruebas son:

- Sistema operativo: Ubuntu 14.04.6 LTS
- Procesador: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
- RAM:
  - RAM 1: Kingston 9905428-417.A00LF 8 GB,
  - RAM 2: SK Hynix HMT451S6BFR8A 4 GB.

5.2. Diseño del mapa

En esta sección se muestra el diseño de los diferentes mapas artificiales en los cuales se prueban los diferentes métodos de optimización.

5.2.1. Mapa cuadrado

Para crear un mapa cuadrado se construye una caja sin tapa o cuatro murallas. Se crea la nube de puntos en el siguiente orden:

1. La primera muralla desde el punto (0.0.0) agregando 0,1 al eje x hasta 5 metros y 0,3 hacia arriba hasta 3 metros llegando al punto (5.0.3).

2. La segunda muralla partiendo del (5,0.1,3) agregando 0,1 metros en el eje y y 0,3 hacia arriba hasta 3 metros llegando al punto (5.0.3).
3. La tercera muralla parte en (4,9.5.0) y se le resta 0,1 metros al eje x hasta el punto (0.5.3)
4. Finalmente, la cuarta muralla que parte (0.4,9.0) restando 0,1 metros al eje y llegando al punto (0.0,1.3).

Como se ve en la siguiente figura 5.1 y en la figura 5.2 con vista superior.

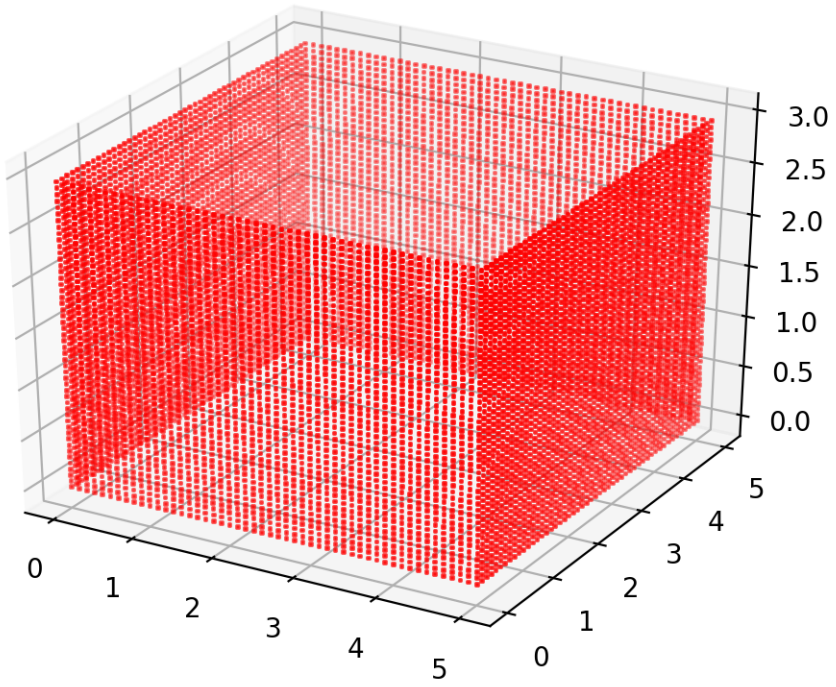


Figura 5.1: Ejemplo de espacio vista frontal

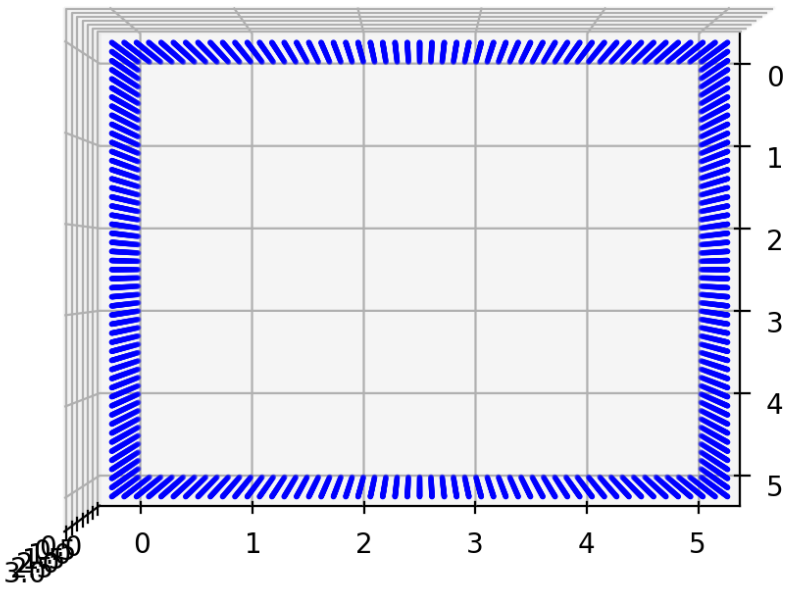


Figura 5.2: Ejemplo de mapa con vista superior

Por lo cual sumando 1 al índice se tiene el punto arriba (siguiente en el eje z) y si le suma la cantidad de puntos y la cantidad de puntos en una columna (que esta guardado durante construcción) da el siguiente punto en el eje *x-y*) se puede calcular los vecinos inmediatos.

5.2.2. Mapa en forma de L

El segundo mapa se diseña con el concepto de una esquina de un pasillo, como se ve en la figura 5.3.

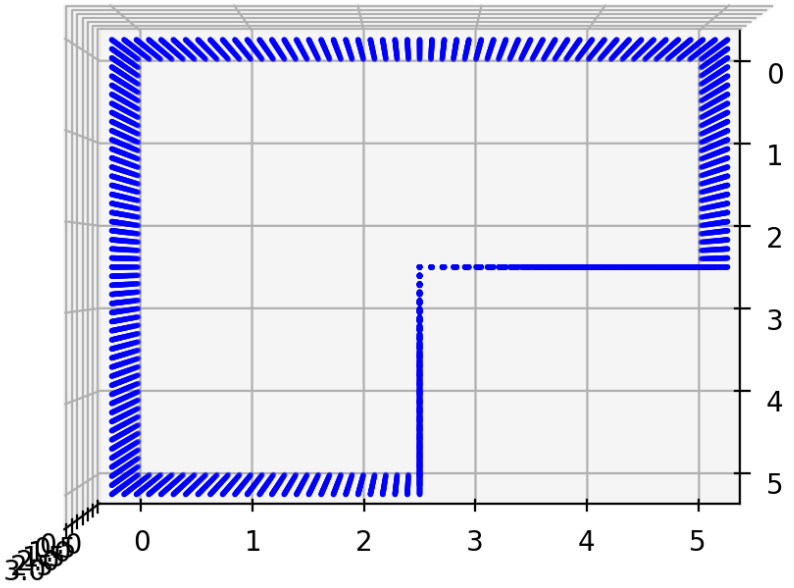


Figura 5.3: Ejemplo de mapa con forma de L, con vista superior

5.2.3. Mapa en forma de H

El tercer mapa se diseña con la característica de una puerta entre 2 habitaciones, parecida a una forma de una H, como se ve en la figura 5.4.

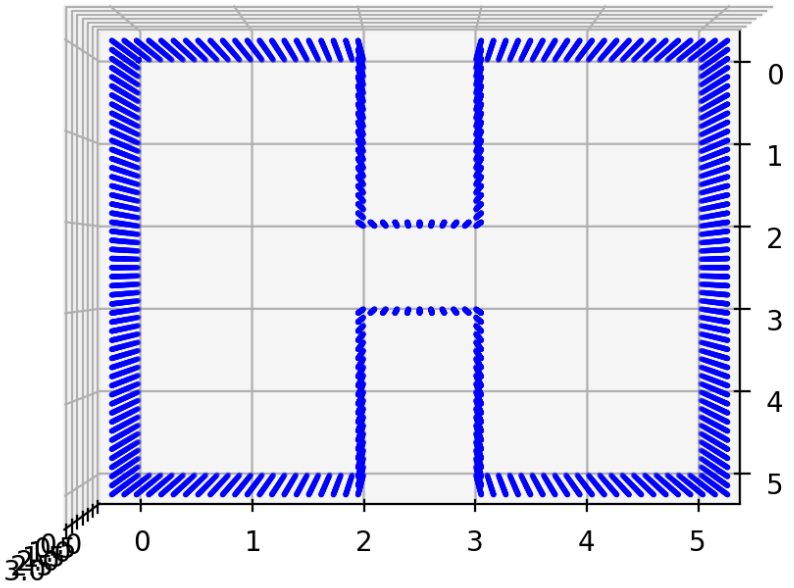


Figura 5.4: Ejemplo de mapa con forma de H, con vista superior

5.3. Resultados en los diferentes mapas

Se presenta la cantidad de iteraciones necesarias para lograr 25 %, 50 %, 75 %, 100 % de puntos vistos por el dron por cada mapa y por cada método probado como explicado en la sección 2.2.3. Los resultados de los algoritmos que no completan la revisión del mapa entero se presentan en el anexo A.1.

5.3.1. Mapa cuadrado

Usando el mapa cuadrado representado en la figura 5.2 los resultados de los algoritmos exitosos se presentan en la Tabla 5.1.

Porcentaje del mapa visto [%]	Powell	COBYLA	Nelder-Mead
25	8	18	17
50	28	33	35
75	44	68	61
100	169	192	330

Tabla 5.1: Cantidad de iteraciones por método, en el mapa cuadrado, para lograr el porcentaje señalado de puntos del mapa

5.3.2. Mapa en forma de esquina

Usando el mapa en forma de esquina representado en la figura 5.3 los resultados de los algoritmos exitosos se presentan en la tabla 5.2.

Porcentaje del mapa visto [%]	Powell	COBYLA	Nelder-Mead
25	8	10	16
50	21	33	41
75	36	59	56
100	185	185	315

Tabla 5.2: Cantidad de iteraciones por método, en el mapa L, para lograr el porcentaje señalado de puntos del mapa

5.3.3. Mapa con forma de H

Usando el mapa en forma de H representado en la figura 5.4 los resultados de los algoritmos exitosos se presentan en la tabla 5.3.

Porcentaje del mapa visto [%]	Powell	COBYLA	Nelder-Mead
25	12	17	22
50	29	44	55
75	53	86	112
100	199	300	>500

Tabla 5.3: Cantidad de iteraciones por método, en el mapa H, para lograr el porcentaje señalado de puntos del mapa

5.4. Análisis y Discusión

Tal como lo revisado en la sección 5.3 de los resultados, el método Powell con su búsqueda del mínimo con respecto al sistema de coordenada, el algoritmo Powell logra las menores cantidades de ciclos en los tres mapas en los diferentes porcentajes, como se ve en las Tablas 5.1, 5.2 y 5.3. En el mapa cuadrado logra revisar todo el mapa en 169 iteraciones en comparación con COBYLA y Nelder-Mead se demoran 192 y 330 iteraciones respectivamente.

Aumentando la cantidad de puntos o la cantidad de murallas no muestra un cambio significativo en la cantidad de ciclos para el método Powell, como se muestra en el Gráfico 5.5.

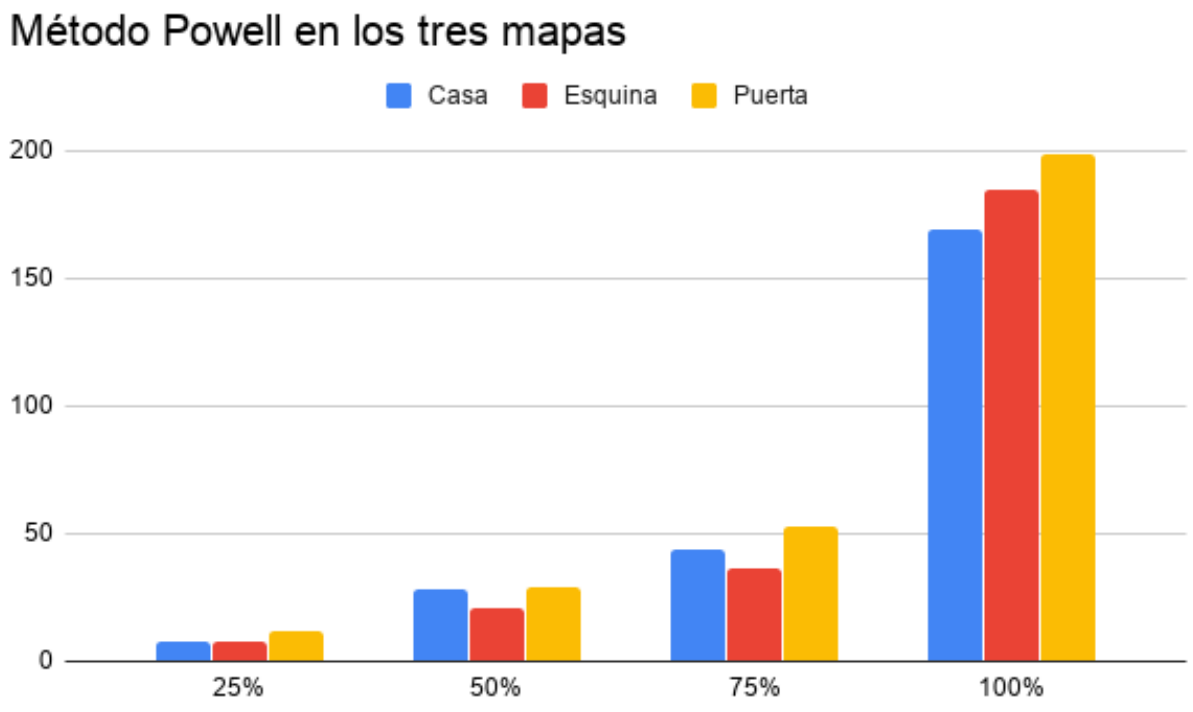


Figura 5.5: Gráfico comparativo de método Powell en los tres mapas cantidad de ciclos contra porcentajes

Se considera que el 75 % del mapa es una buena representación de la cantidad de puntos vistos, dado que los puntos restantes se encuentran aislados y dispersos y tomaría muchos ciclos revisarlos, como se puede ver en la Figura 5.5, donde el 75 % de los puntos toma 50 ciclos y lograr el 100 % toma 170 ciclos aproximadamente.

En la figura 5.6 se muestra la secuencia de poses determinadas por el método Powell.

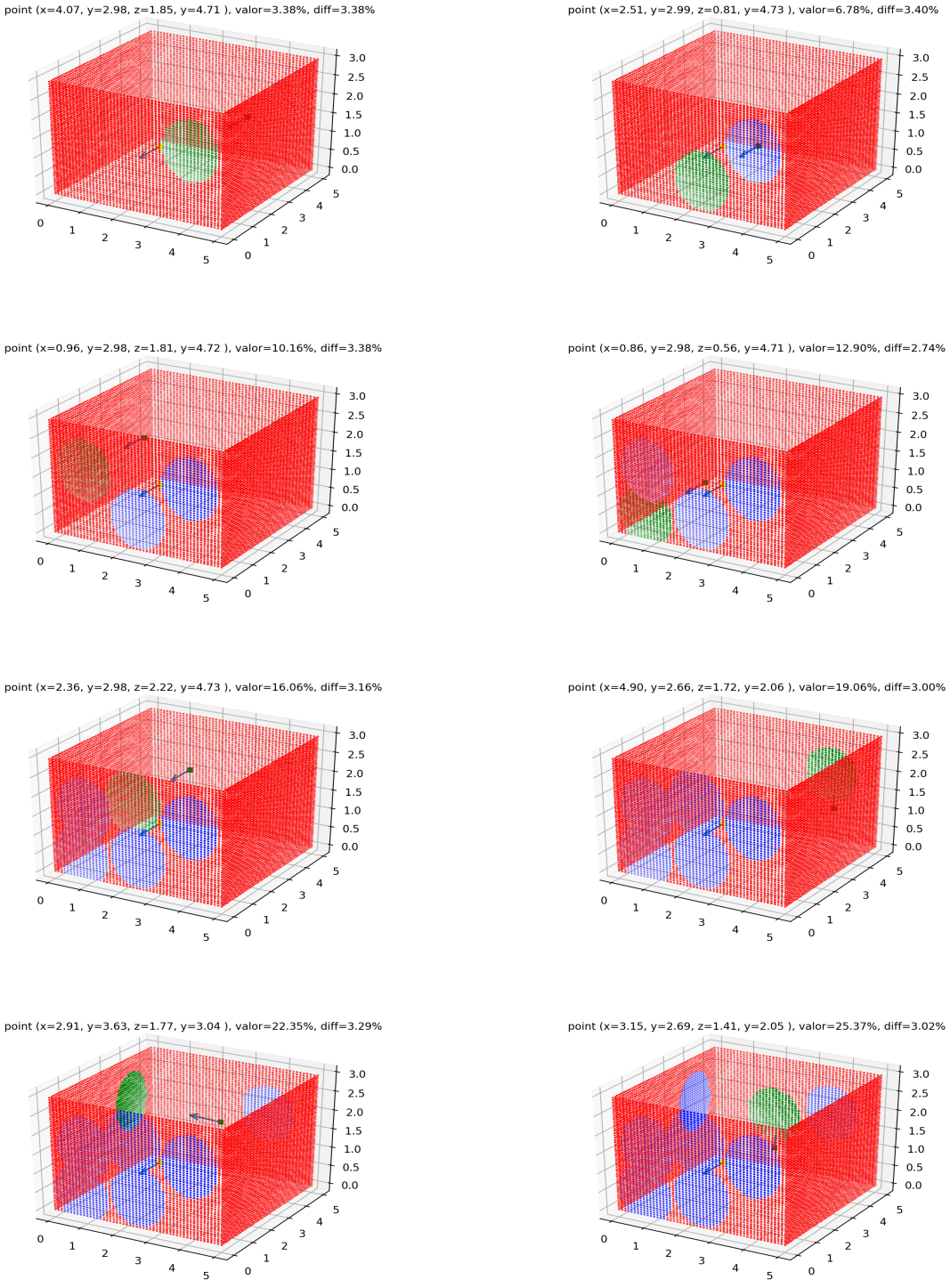


Figura 5.6: Secuencia de poses con el método Powell

En la figura 5.7 se muestra la secuencia de poses determinadas por el método Nelder–Mead.

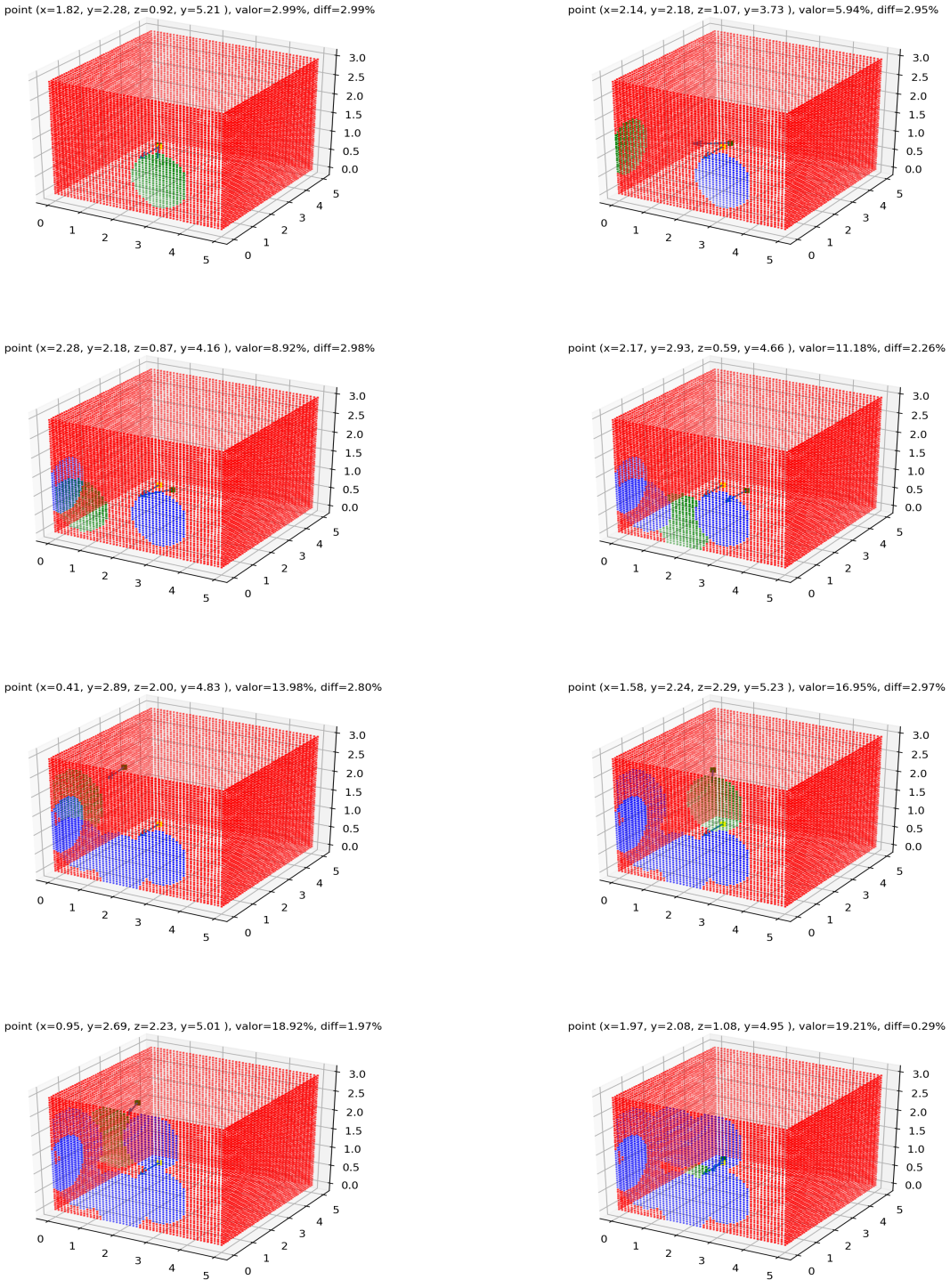


Figura 5.7: Secuencia de poses con el método Nelder–Mead

Capítulo 6

Conclusión y Trabajo Futuro

Durante el desarrollo de esta tesis se implementaron los algoritmos necesarios para:

- La simulación de un dron.
- La planeación y seguimiento de la trayectoria.
- La revisión del mapa por los métodos de optimización y medición del proceso.

Con estos algoritmos se probaron los diferentes métodos de optimización en los tres mapas.

Usando el mapa definido se implementa un algoritmo que permite calcular la próxima pose usando el *framework* SciPy, que itera determinando la pose que minimiza la cantidad de puntos no visto, hasta que el trayecto a visto todos los puntos, revisada en la sección 4.3. Para representar el mapa del ambiente se usa el modelo de una nube de puntos (x,y,z) con una referencia booleana de si el punto fue "visto", conectados por una triangulación de Delaunay. Como métrica de desempeño del algoritmo de planificación de trayectoria, se usa la proporción de la cantidad de puntos vistos contra la cantidad de puntos totales.

Para hacer seguimiento de trayectoria se implementa el algoritmo que, sigue una lista de poses dada. Se implementa en dos etapas de control, primero un sistema que: maneja y guarda las poses y envía la diferencia entre la pose actual y la objetivo a la segunda fase, que se encarga de arribar a la pose a través de un controlador PID.

Como casos de prueba se diseña e implementa tres mapas distintos, estos tratan de representar zonas que se pueden encontrar en interiores, donde se midieron los diferentes métodos de optimización.

Cuando se implementa el sistema de optimización de poses, se encontró que ningún método funcionaba solo ofreciéndole poses establecidas; se necesitó implementar una función llamada *recommend_pose* , ver sección 4.4.4, que le suministraba una pose que utiliza la información del mapa para determinar poses que puedan "ver" puntos del mapa sin ver.

Se confirma la hipótesis, de que existe un método de optimización que logra generar un mapa de tres dimensiones. Dado que a pesar de las simplificaciones que se realizaron al problema, se demuestra que el método de optimización Powell, a pesar de no estar diseñado para mapas discon-

tinuos, logra con ayuda de la función *recommend_pose*, revisar todos los puntos de la nube.

Si suponemos una relación entre la complejidad del mapa y la cantidad de puntos en el mapa se estima que el método Powell podría funcionar en mapas complejos, en general manejar la cantidad de puntos del mapa y la elección de puntos representativos son variables que controlar en caso de mapas complejos tanto para lograr revisar el mapa como cuanto tiempo se demore.

Dado las simplificaciones del problema para el algoritmo se requieren implementar varios módulos que le permitan autonomía en una habitación real, algunos de estos módulos son: evasión de obstáculos tanto durante el planeo del camino como durante el vuelo, modificación del mapa usando los datos obtenidos del sonar y la cámara, sistema de advertencia de baja batería o problemas de motor y otros módulos necesarios son mencionados en el trabajo futuro.

6.1. Resultados Obtenidos

Se teoriza que el método Powell presenta mejores resultados gracias a que no usa información de la derivada de la función. Los valores sobre derivadas de la función no son útiles en este caso por dos razones:

- El mapa es discreto por lo cual no es derivable. La función es escalonada, por lo cual la derivada es la función delta de Dirac.
- Dado que el mapa es discreto o la función a optimizar es escalonada, si se calcula la derivada en un punto es cero o infinito.

Al comparar las diferentes secuencia de poses entre Powell y Nelder–Mead en las figuras 5.6 5.7, se puede notar que a pesar de que Nelder–Mead logra pasar a otra muralla en la segunda pose, Powell logra revisar mayor cantidad de puntos en la primera muralla antes de cambiar a otra. Se puede suponer que esta es una estrategia deseable.

Se remarca que la constante proporcional del PID es el cual afectan mayormente el vuelo del dron, como se ve en la sección 4.2.3.

6.2. Trabajo futuro

Se nombran varios puntos que se pueden mejorar en el algoritmo.

- Probar el software en un dron real. La prueba de este software y los diferentes algoritmos de optimización en un ambiente real puede mostrar resultados interesantes.
- Reescribir el código para permitir diferentes formas del dron o diferentes robots. El controlador se puede extraer de su propio archivo y utilizar un adaptador, para el servidor de *waypoints* se podría generalizar la cantidad de dimensiones, y odometría interna y QR tiene que cambiarse dependiendo de los sensores.

- Agregar un sistema automático de determinación de normales. Generar manualmente un nuevo ambiente es complejo de por sí, uno de los requerimientos del mapa es contener normales, por lo cual generar automáticamente las normales simplificaría crear mapas.
- Mejorar el sistema de definición del mundo. Uso de diferentes modelos del mundo puede mejorar el rendimiento del algoritmo y el posterior uso del modelo.
- Optimizar la función de visión de cono, el algoritmo recorre todos los puntos del mapa lo cual es innecesario, se puede ahorrar tiempo y recursos durante el planeo de trayectoria.
- Optimizar la función de recomendación de pose; el algoritmo recorre todos los puntos del mapa lo cual es innecesario. Se puede ahorrar tiempo y recursos durante el planeo de trayectoria.
- Agregar un nodo para modificar el modelo del mundo usando los sensores del dron luego de cada iteración. Actualmente el mapa no se actualiza después de que el dron arriba a una pose, esto es importante para que el software realice SLAM.
- Comparar métodos de optimización a diferentes niveles de granularidad del mapa. Se sospecha que existe un punto de densidad del mapa donde el cálculo de las derivadas se vuelve viable, para probar esto se debería diseñar e implementar experimentos que comparen la capacidad de revisar todo el mapa y el tiempo que le tome en niveles de granularidad.
- Mejorar el código con el objetivo de agregar fácilmente nuevos códigos de optimización con menor modificación. El uso de patrones de diseño de software, por ejemplo, *factory method* mejoraría la calidad y la facilidad de probar nuevos métodos de optimización.
- Agregar módulo de prevención de choques. Se puede implementar un sistema de prevención de choques usando la información disponible en el mapa durante el planeo de trayectoria y durante el vuelo.
- Diseñar métricas de optimización. Diferentes métricas de optimización pude revelar debilidades en los métodos. Un ejemplo de métrica es la cantidad de veces que un punto es revisado múltiples veces.

Aunque lo principal como trabajo futuro debe ser la implementación de diferentes algoritmos para encontrar el camino óptimo y las mediciones de la incertidumbre del mapa.

Se recomienda hacer métodos de optimización que usen la menor cantidad de información de la derivada o diseñar e implementar funciones que provean la información de las derivadas dependiendo del mapa.

Bibliografía

[1] Delaunay triangulation. https://es.mathworks.com/help/matlab/math/delaunay-triangulation.html. Accesado: 2019-09-13.

[2] Ejes del avión. https://www.researchgate.net/profile/Sriharsha_Etigowni/publication/329521700/figure/fig2/AS:701942607138816@1544367567309/Drones-pitch-roll-and-yaw.png. Accesado: 2010-09-30.

[3] Parrot ar.drone 2.0. https://www.parrot.com/files/s3fs-public/styles/product_teaser_hightlight/public/ar_drone_power_edition_orange.png?itok=mMioXD5X. Accesado: 2010-09-30.

[4] Pid. https://commons.wikimedia.org/wiki/File:PID_en_updated_feedback.svg. Accesado: 2019-11-22.

[5] Slsqp. https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/#slsqp. Accesado: 2019-10-17.

[6] R. Bajcsy. Active perception. *Proceedings of the IEEE*, 76(8):966–1005, Aug 1988.

[7] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on Robotics*, 32(6):1309–1332, Dec 2016.

[8] Henry Carrillo, Philip Dames, Vijay Kumar, and José A Castellanos. Autonomous robotic exploration using occupancy grid maps and graph slam based on shannon and rényi entropy. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 487–494. IEEE, 2015.

[9] Gabriele Costante, Christian Forster, Jeffrey Delmerico, Paolo Valigi, and Davide Scaramuzza. Perception-aware path planning. *arXiv preprint arXiv:1605.04151*, 2016.

[10] Ayoung Kim and Ryan M. Eustice. Active visual slam for robotic area coverage: Theory and experiment. *The International Journal of Robotics Research*, 34(4-5):457–475, 2015.

[11] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, Sep. 2004.

[12] Jorge Nocedal and Stephen J Wright. Sequential quadratic programming. *Numerical optimization*, pages 529–562, 2006.

[13] Michael JD Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis*, pages 51–67. Springer, 1994.

[14] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

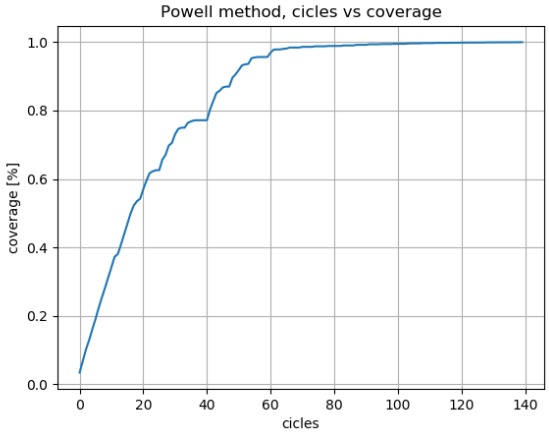
[15] Quirón5. Ejes del avión. `https://upload.wikimedia.org/wikipedia/commons/3/32/Avión mov3.JPG`. Accesado: 2010-09-30.

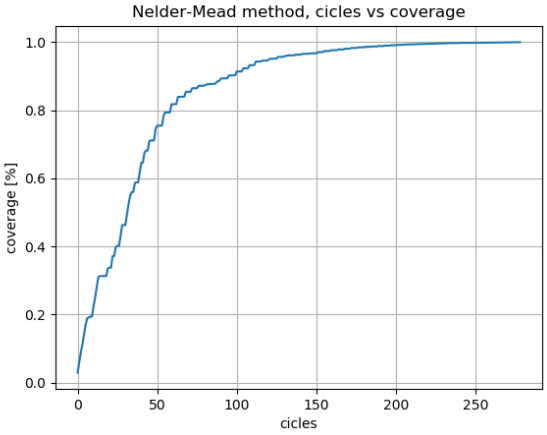
[16] Sebastian Thrun, Yufeng Liu, Daphne Koller, Andrew Y Ng, Zoubin Ghahramani, and Hugh Durrant-Whyte. Simultaneous localization and mapping with sparse extended information filters. *The international journal of robotics research*, 23(7-8):693–716, 2004.

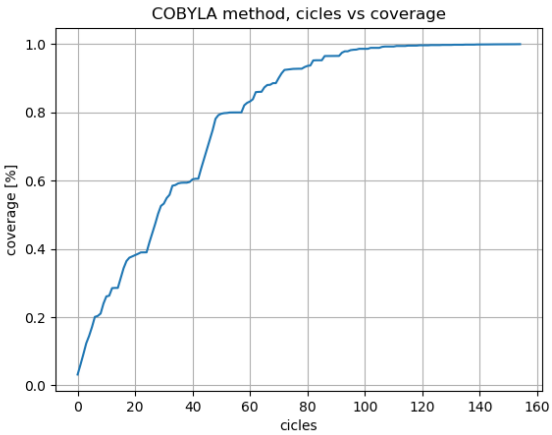
Apéndice A

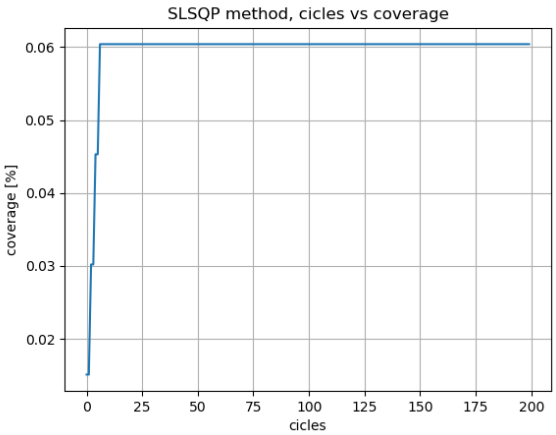
Anexo

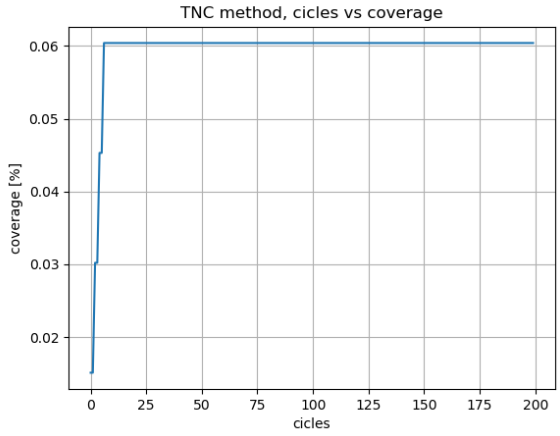
A.1. Gráficos de mapa cuadrado

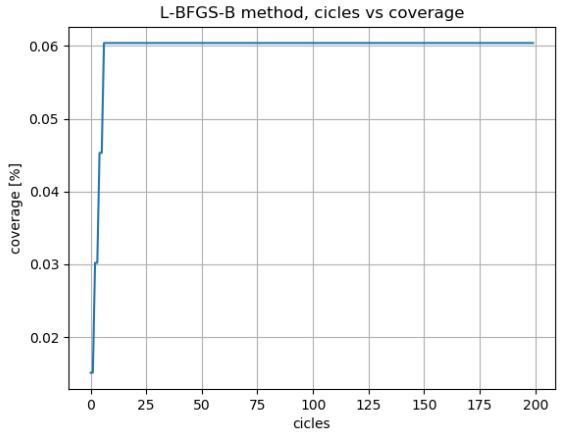


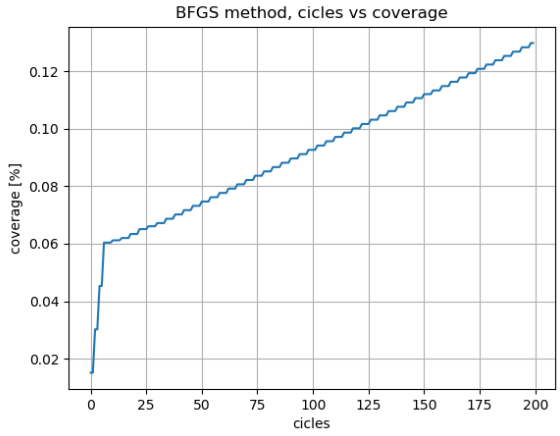


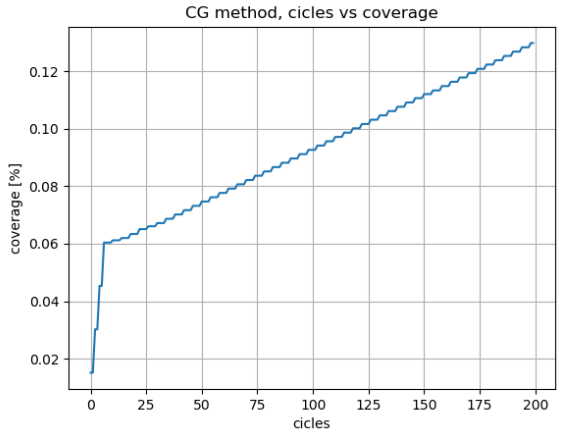




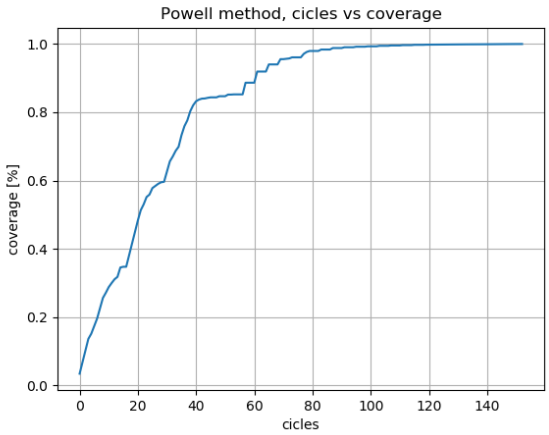


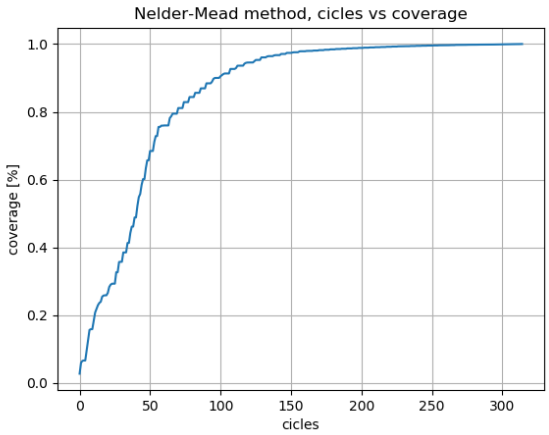


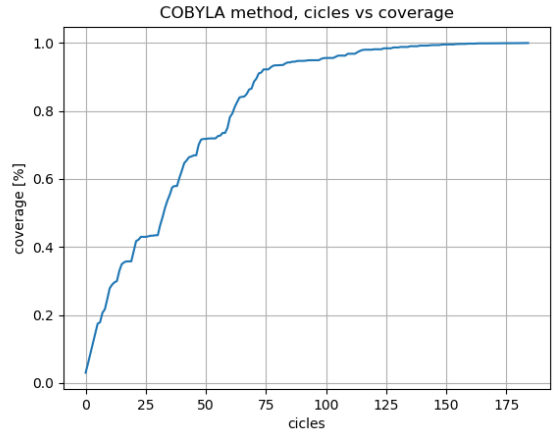


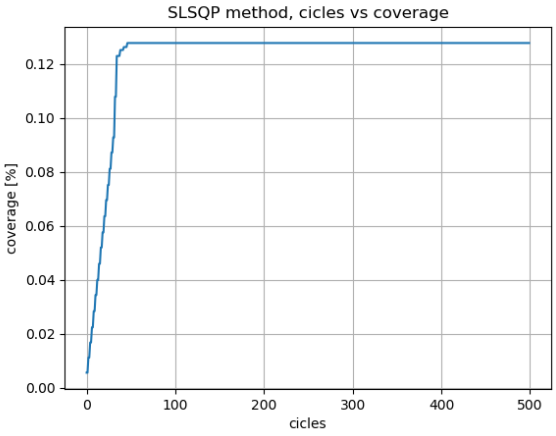


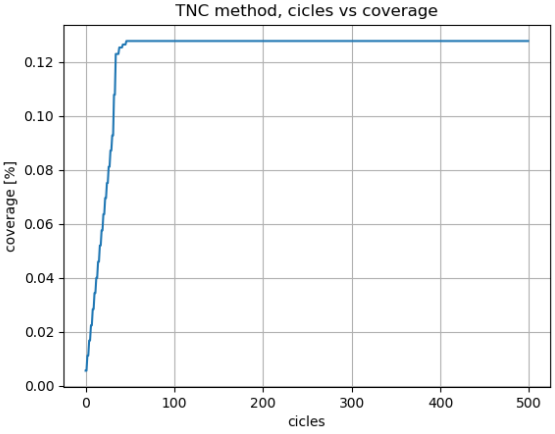
A.2. Gráficos de mapa con forma de L

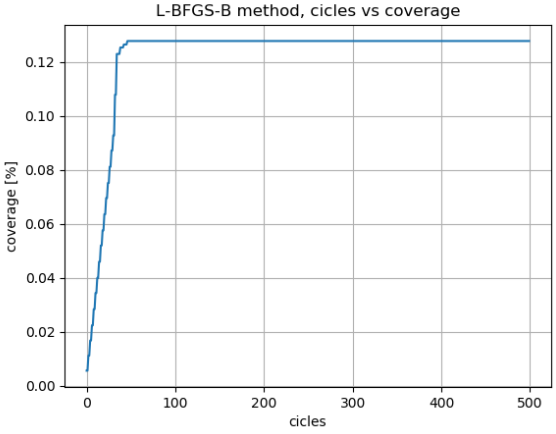


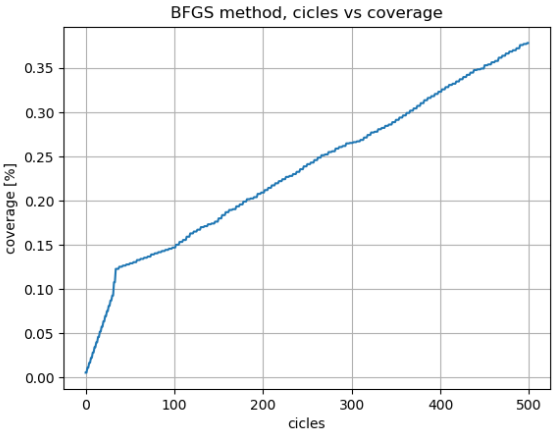


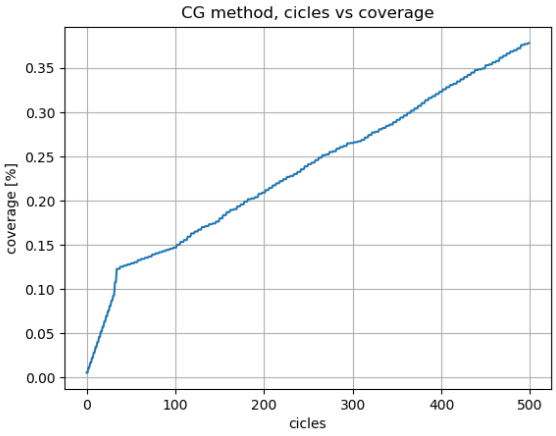

COBYLA method, cicles vs coverage
coverage [%]
cicles
0.0
0.2
0.4
0.6
0.8
1.0
0
25
50
75
100
125
150
175


SLSQP method, cicles vs coverage
coverage [%]
cicles
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
100
200
300
400
500

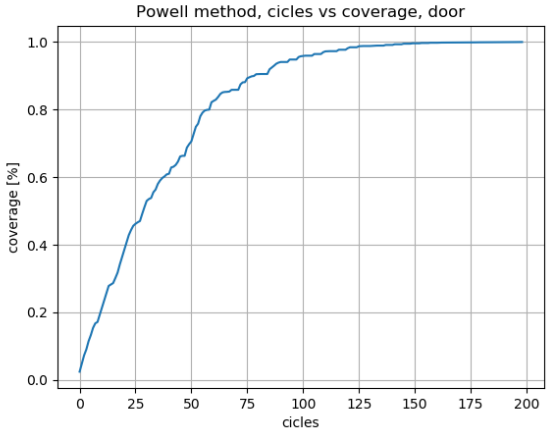

TNC method, cicles vs coverage
coverage [%]
cicles
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
100
200
300
400
500

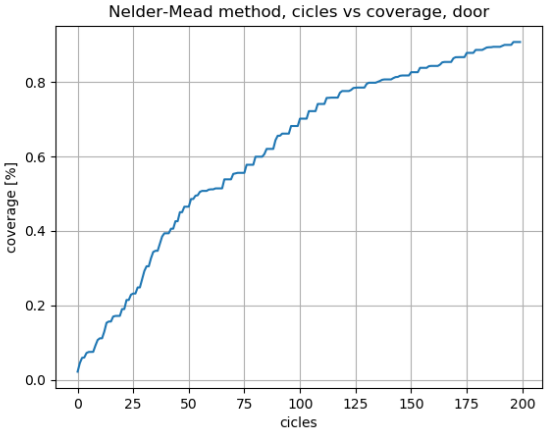

L-BFGS-B method, cicles vs coverage
coverage [%]
cicles
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
100
200
300
400
500

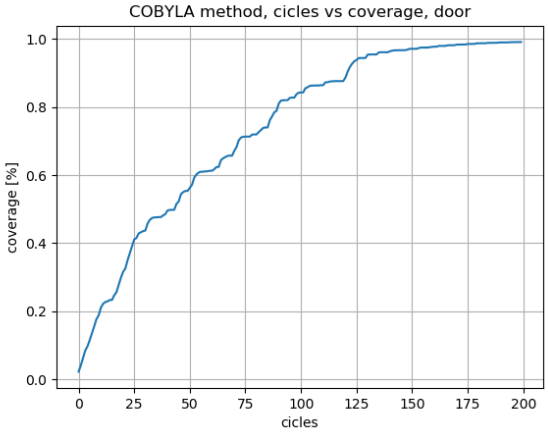

BFGS method, cicles vs coverage
coverage [%]
cicles
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
100
200
300
400
500

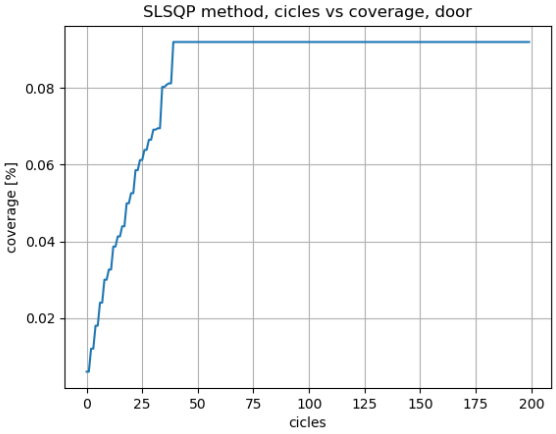

CG method, cicles vs coverage
coverage [%]
cicles
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
100
200
300
400
500

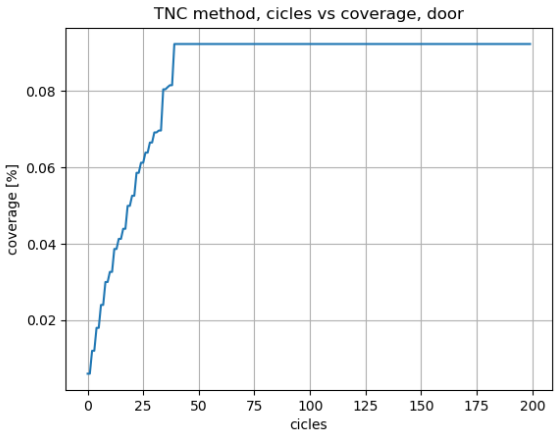
A.3. Gráficos de mapa con forma de H

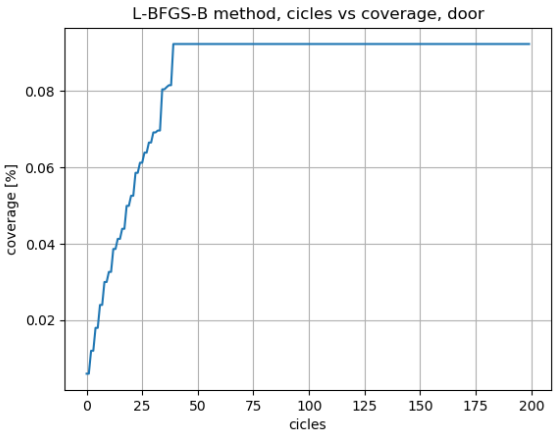


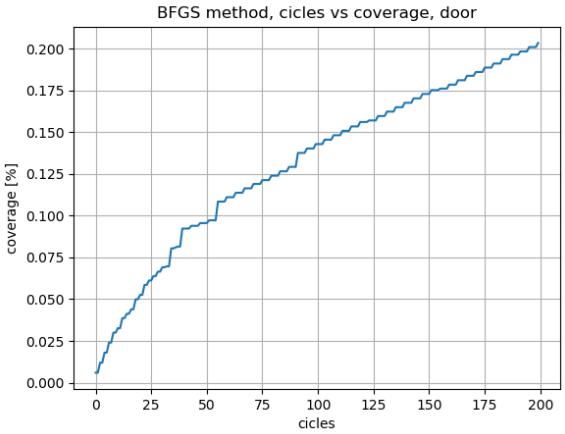


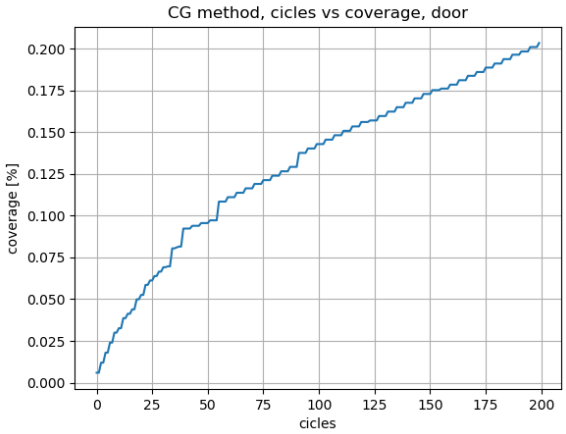









BFGS method, cicles vs coverage, door
coverage [%]
cicles
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0
25
50
75
100
125
150
175
200


CG method, cicles vs coverage, door
coverage [%]
cicles
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0
25
50
75
100
125
150
175
200