NAVEGACIÓN AUTÓNOMA DE UN QUADROTOR EN ENTORNOS INTERIORES MEDIANTE SLAM MONOCULAR

Adriana Paola Sarmiento Andrade

Diego Alejandro Ramírez Patiño

Departamento de Electrónica, Facultad de Ingeniería

Pontificia Universidad Javeriana

2014

AGRADECIMIENTOS

Adriana Paola

Agradezco a Dios por llenarme de sabiduría y bendiciones, además de llenarme de fortaleza espiritual para afrontar el gran reto que represento este trabajo.

A mis papás, Adriana y Luis Eduardo, por sus palabras de apoyo en cada momento de dificultad a lo largo de toda mi carrera, por ese apoyo incondicional lleno de enseñanzas cada día y a mi hermano, Andrés, por sus grandes consejos, su tranquilidad y serenidad y por ser un gran ejemplo a seguir en mi vida. A mi compañero Diego, por ser no solo un compañero de estudio si no ser ese apoyo en todo aspecto de mi vida desde hace 3 años, por su dedicación, responsabilidad y entrega en todo el desarrollo de este trabajo pero sobre todo por ser esa gran persona llena de enseñanzas y fortaleza que se encargó de hacerme ver la luz al final del túnel cuando sentía que no iba a poder terminar este proyecto.

Al ing. Julián Colorado por la confianza que nos dio para desarrollar este proyecto, por su acompañamiento y preocupación para tener un feliz término.

Quiero dar el más grande agradecimiento a Johana Flórez y a Francisco, que sin ninguna obligación nos prestaron su ayuda en cada momento que lo necesitamos, por sus consejos, paciencia y por sus palabras de apoyo y aliento en esos momentos de desespero.

Diego Alejandro

Siendo este el reto más desafiante de mi vida académica, quisiera agradecer a todas aquellas personas que han permanecido siempre como un gran apoyo. Primero que todo quiero agradecer a Dios por darme la familia que tengo, ya que gracias a ellos este proyecto ha podido salir adelante a pesar de muchas dificultades que se tuvieron durante el camino.

Gracias a mis papás Diana y Ricado, hermanos Karen y Andrés Felipe que han sido parte fundamental de mi vida, gracias por apoyarme, brindarme consejos y enseñarme cosas nuevas todos los días. Gracias a mi gran compañera de trabajo Paola, por todos los aportes que ha hecho en mi vida personal y académica, por intentar hacer de mí una mejor persona durante estos últimos tres años y porque definitivamente sin ella nada de esto habría sido posible; su tenacidad, constancia y amor han hecho de este tiempo difícil un tiempo de gran aprendizaje.

Gracias al Ing. Julián Colorado por confiar en nuestras capacidades para el desarrollo de este trabajo de grado y por todas y cada una de sus sugerencias y correcciones.

Finalmente, quisiera hacer una mención especial a Johana Flórez y Francisco, quienes fueron parte de la columna vertebral de este trabajo de grado, ya que con su conocimiento apoyaron de manera permanente el desarrollo de este proyecto y definitivamente son a quienes debo dar un gran reconocimiento a su paciencia por el sin número de horas que de manera muy amable nos regalaron.

Tabla de contenido

1. INTRODUCCIÓN

En robótica, uno de los grandes problemas a solucionar es la auto-localización o ubicación autónoma de un robot en el espacio, dado que éstos generalmente estarán funcionando en entornos desconocidos, es decir, entornos que no vienen precargados de alguna manera en su memoria.

La forma más común de localización y auto-localización a nivel global, está dada en coordenadas que pueden ser proporcionadas por un GPS (Global Position System), sin embargo hay situaciones específicas en las cuales no es posible hacer uso de este tipo de sistemas, por ejemplo, el posicionamiento en entornos interiores (dentro de un edificio), donde por cuestiones de conectividad, el GPS no puede determinar las coordenadas. Es por esto que se han venido estudiando diferentes técnicas de auto-localización que no dependan del uso de un GPS [1].

Una de las técnicas más utilizadas en el mapeo de entornos interiores se conoce como *Simultaneous Localization and Mapping*, SLAM por sus siglas en inglés. Esta técnica investiga diferentes formas que permiten al robot tener una ubicación con respecto al entorno en el que se está moviendo, haciendo uso de sensores que le permiten percibir cierto tipo de características de dicho entorno. Entre los sensores más usados se encuentran las cámaras, unidades inerciales y lásers. Dentro del primer grupo mencionado, cabe destacar que pueden ser usadas una (visión monocular) o varias cámaras [2] (visión estereoscópica) que permiten tener redundancia en los datos, pero a su vez exigen un hardware que sea capaz de procesar los datos y adicionalmente tener el espacio suficiente en el robot para la cantidad de cámaras que se deseen usar.

Actualmente se han desarrollado diferentes tipos de robots cuya morfología varía completamente dependiendo de la aplicación. Existen dos grandes grupos de robots: terrestres [3] y aéreos. Adicionalmente ya se encuentran en el mercado algunos ejemplares de bajo costo considerados robots para diversión, muchos de ellos equipados con cámaras, unidades inerciales, hélices, alas, entre otras características.

Uno de los robots de bajo costo mencionados anteriormente es el quadrotor AR.drone 2.0 [4], usado comercialmente para tomar fotografías mientras se encuentra en el aire, donde el usuario puede controlar mediante su celular la altitud, velocidad y el instante en el cual quiere tomar una fotografía, la cual quedará guardada inmediatamente en su teléfono móvil.

La forma de conectarse y comunicarse con el AR.drone 2.0 se realiza mediante Wi-Fi. El AR.drone 2.0 publica una red inalámbrica a la cual debe conectarse la persona que quiera controlarlo y mediante aplicaciones gratuitas desarrolladas por Parrot (fabricante del quadrotor AR.drone) enviarle comandos de una manera sencilla e intuitiva para el usuario.

Dentro de las técnicas de SLAM hay varios problemas que deben ser resueltos de maneras diferentes, teniendo en cuenta las características y sensores del robot. En nuestro caso, se hará uso únicamente de la cámara frontal con la que viene equipado el quadrotor AR.drone 2.0.

Específicamente cuando se emplea una sola cámara para realizar el proceso de SLAM, se le conoce como SLAM monocular. Esta técnica ha tomado gran fuerza en los últimos años, ya que permite el mapeado de un entorno desconocido mediante la identificación de marcas visuales, logrando reconstruir de manera digital la forma en la cual el robot se está moviendo con relación a su entorno.

Los principales problemas que deben solucionarse en una técnica de SLAM monocular son la estimación de la profundidad de los objetos y características, teniendo en cuenta que solo se cuenta con una sola cámara para realizar este proceso.

En este documento se detallará la integración y evaluación de una técnica de SLAM monocular basada en características (en este caso códigos QR a los cuales se hará referencia como códigos de aca en adelante) en el quadrotor AR.drone 2.0 para entornos interiores. Adicionalmente se realizará la explicación de la creación de un entorno de simulación en Gazebo, el cual es un software especializado para este tipo de requerimientos.

En la primera parte de este documento el lector encontrará una completa descripción de los conceptos necesarios para entender de manera apropiada los temas relacionados con visión computarizada y funciones que ayudaron en el proceso de implementación del algoritmo de slam monocular.

Posteriormente se explican las limitaciones consideradas para el desarrollo de este proyecto, teniendo en cuenta las restricciones hardware del quadrotor de bajo costo utilizado.

Luego se explica de manera detallada los algoritmos desarrollados mediante una sencilla descripción del seudocódigo de cada uno de ellos, dejando claro el objetivo de cada algoritmo y las posibles mejoras que podrían llevarse a cabo para mejorar su desempeño. Por otro lado se detalla el procedimiento seguido para la creación del entorno de simulación.

Finalmente se realizaron pruebas experimentales y se muestran los resultados con el objetivo de medir la precisión de la técnica de SLAM implementada en un robot de bajo costo, el cual podría ser utilizado en labores de inspección y rescate en entornos interiores.

2. MARCO TEÓRICO

Actualmente hay una tendencia en el uso de Vehículos Aéreos no Tripulados (por su siglas en inglés, UAV) en áreas como la agricultura [5], búsqueda-rescate [6], aplicaciones militares [7], vigilancia-supervisión, y fotografía aérea para levantamiento topográfico, etc. Dentro de la amplia categoría de los UAV, los quadrotors son plataformas de 6 grados de libertad (GdL) con características muy peculiares que hacen de esta plataforma un sistema idóneo para la navegación en entornos interiores de difícil maniobrabilidad. Entre estas características se encuentran: estabilidad en vuelo gracias a sus cuatro rotores simétricos, capacidad de vuelo estacionario, capacidad de vuelo vertical (VTOL) y control de orientación alrededor de su propio eje.

2.1 ROS

Robot Operating System, ROS [8] por sus siglas en inglés, es un conjunto de librerías que permiten el desarrollo de código compacto y compatible con diferentes arquitecturas hardware de robots. Entre sus funcionanlidades, ROS permite la comunicación bidireccional entre un robot y una estación base, así como la gestión y procesamiento de la información. En el desarrollo de este trabajo se hará uso de un conjunto de funciones llamadas 'nodos' [9] [10] [11], las cuales permiten la teleoperación y obtención de datos en tiempo real relacionados con el vuelo del robot.

El resultado de este trabajo concluye con la creación de un paquete de ROS escrito en el lenguaje de programación C++, que puede ser ejecutado de manera sencilla en cualquier estación de trabajo que cuente con este conjunto de librerías. Este paquete extiende la funcionalidad del robot en términos de navegación por medio de una técnica de SLAM.

2.2 OPENCV

Open Source Computer Vision Library, OpenCV [12] por sus siglas en inglés, es un conjunto de librerías de código abierto usado para el tratamiento de imágenes en visión computarizada. OpenCV fue desarrollado con el fin de proveer una infraestructura común para aplicaciones de visión mediante computador, y de esta manera acelerar el desarrollo de productos comerciales que incluyan percepción por medio de imágenes.

Debido a lo anterior se seleccionó este conjunto de librerías para el tratamiento de las imágenes recibidas de la cámara frontal del AR.drone 2.0. Adicionalmente es importante resaltar que existe una alta compatibilidad entre OpenCV y ROS, lo cual nos proporciona un entorno ideal para el desarrollo de nuestro trabajo.

2.2.1 MODELO PINHOLE

El modelo pinhole [13] hace una descripción matemática de la relación existente entre un punto en coordenadas 3D y su proyección en el plano de la imagen de una cámara pinhole ideal, donde la apertura de la cámara está descrita como un punto y se hace la suposición que no se están usando lentes para concentrar la luz.

Es importante destacar que este modelo no incluye distorsiones geométricas ni desenfoques debidos a lentes y aperturas de tamaño finito. Este modelo es simplemente una primera aproximación del mapeado

de una escena en 3D a una imagen 2D. La validez del modelo depende directamente de la calidad de la cámara y en general decrece desde el centro de la imagen hasta los bordes en la medida en que incrementa el efecto de distorsión del lente.

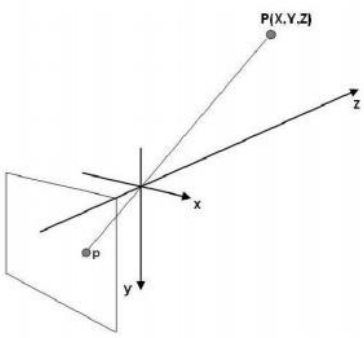


Figura 1. Geometría de una cámara pinhole

En la figura 1 se muestra la geometría de una cámara pinhole, donde se puede observar que se tiene un sistema coordenado en tres dimensiones, el cual está ubicado justo donde se encuentra la apertura de la cámara. El eje z de este sistema está apuntando en la dirección de visión de la cámara y se conoce como el eje óptico o eje principal.

El plano de la imagen es el lugar donde el mundo 3D es proyectado a través de la apertura de la cámara. El plano de la imagen es paralelo a los ejes '*x'* y '*y'*, el cual está localizado a una distancia *f* (distancia focal) del origen del sistema coordenado 3D en dirección negativa del eje z.

El punto P está ubicado en algún lugar del mundo con unas coordenadas X, Y, Z relativo al eje 'x', 'y', 'z'. La línea de proyección del punto P al plano de la imagen es la línea que cruza por el origen del sistema coordenado y la proyección de dicho punto se denota en la imagen como p, el cual en este plano también cuenta con un sistema coordenado en 2D.

La apertura de la cámara pinhole, a través del cual deben pasar todas las líneas de proyección, es asumida muy pequeña, tanto como un punto y se conoce como el centro óptico de la cámara.

Ahora se desea saber cómo las coordenadas 2D del plano de la imagen del punto p dependen de las coordenadas X, Y y Z del punto P. En la figura 2, se presenta esta misma escena pero vista lateralmente con el fin de dar una mejor explicación de dicha proyección.

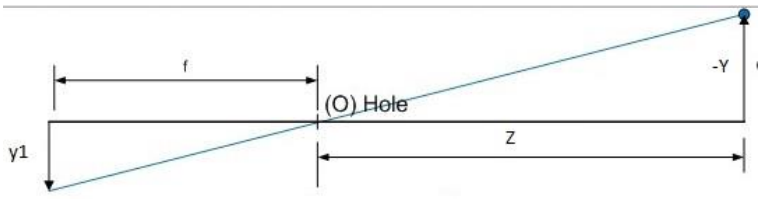


Figura 2. Geometría de la cámara pinhole lateral

En la Figura 2 se pueden observar dos triángulos, ambos tienen parte de la línea de proyección (azul) como su hipotenusa. Los catetos del triángulo de la izquierda son *y1* y *f*, y los catetos del triángulo de la izquierda son –Y y Z.

Al ser dos triángulos similares se tiene que:

$$\frac{Y1}{f} = \frac{-Y}{Z} \qquad (1)$$

Si esta misma situación se observará pero en la dirección negativa del eje X se tiene:

$$\frac{X1}{f} = \frac{-X}{Z} \tag{2}$$

Estas dos expresiones pueden simplificarse, obteniéndose:

$$\begin{pmatrix} Y1 \\ X1 \end{pmatrix} = -\frac{f}{Z}\begin{pmatrix} Y \\ X \end{pmatrix} \tag{3}$$

Esta última expresión describe la relación entre las coordenadas 3D (X, Y, Z) del punto P y las coordenadas de la imagen (X1, Y1) dadas por el punto p en el plano de la imagen.

2.2.2 CALIBRACIÓN DE LA CÁMARA

La necesidad de calibrar una cámara surge como una manera de compensar aquellas deficiencias técnicas durante su proceso de construcción, las cuales se traducen finalmente en distorsiones considerablemente notorias en las imágenes capturadas por estas [14].

Como una alternativa de solución, la calibración de la cámara busca cuantificar las deficiencias en construcción, y basándose en estos valores lograr corregir la imagen con el objetivo de obtener unos datos mucho más cercanos a la realidad cuando se usan procesos de visión computarizada cuyo objetivo requiere de alguna medición o dato preciso a partir de la cámara.

Una cámara tiene parámetros intrínsecos que son aquellos que definen la geometría interna y la óptica de la cámara, dichos parámetros son la distancia focal, el punto principal y el centro óptico que son parámetros constantes, adicionalmente una cámara también tiene parámetros extrínsecos que son aquellos que definen la posición y la orientación del cuadro de referencia de la cámara con respecto al mundo.

Estudiando más de cerca el proceso de calibración, se tiene que para la distorsión se deben tener en cuenta factores radiales y tangenciales.

La distorsión radial produce que líneas rectas dentro de la imagen, se vean como si fueran curvas. Este efecto es más notorio en la medida que nos alejemos desde el centro de la imagen hacia alguno de sus extremos, obteniendo la distorsión máxima en los bordes de la imagen.

El factor de distorsión radial está descrito por la siguiente expresión:

$$Xcorregido = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{4}$$

$$Ycorregido = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{5}$$

Para un pixel ubicado en las coordenadas (x, y) de una imagen afectada por distorsión, se hace una corrección mediante las expresiones anteriores que indican las nuevas coordenadas que debe tener el pixel para obtener una imagen corregida.

La distorsión tangencial por su parte, ocurre debido a que el lente de la cámara con la cual se está capturando la imagen no está alineado de una manera perfectamente paralela al plano de la imagen. El efecto visual que produce este tipo de distorsión, es que algunas áreas dentro de la imagen se ven más cercanas de lo esperado teniendo en cuenta el escenario real capturado.

El factor de distorsión tangencial está descrito por las siguientes expresiones:

$$Xcorregido = x + [2p_1xy\ +\ p_2(r^2 + 2x^2)] \quad (6)$$

$$Ycorregido = y + [p_1(r^2 + 2y^2) + 2p_2xy] \quad (7)$$

Observando los factores de distorsión radial y tangencial, tenemos que existen 5 coeficientes de distorsión:

$$CoeficientesDistorsión = [k_1\ k_2\ p_1\ p_2\ k_3] \quad (8)$$

Todas las expresiones necesarias para la calibración de la cámara nacen de sistemas de ecuaciones que se obtienen a partir de procesos geométricos y proyectivos que pretenden corregir todas las distorsiones que puedan existir en una imagen.

Por otro lado, se tiene una matriz que contiene los parámetros intrínsecos de la cámara y cumple con la siguiente relación:

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (9)$$

Para la expresión anterior w=Z. Los parámetros fx y fy son las distancias focales respecto a cada eje coordenado y cx, cy los centros ópticos expresados en píxeles.

OpenCV cuenta con una función que permite realizar una calibración completa de la cámara a partir de una serie de imágenes de un patrón conocido. Cada una de estas imágenes debe contener una vista diferente del patrón de calibración variando el ángulo desde el cual es capturada la imagen.

Existen varios tipos de patrones de calibración, sin embargo los más conocidos son el tablero de ajedrez y el patrón de círculos. El más utilizado es el tablero de ajedrez, el cual tiene que tener un tamaño mínimo de 4 x 4 cuadros.

Adicionalmente, ROS integra un sistema de calibración de cámara propio de uno de los paquetes que se ejecutan para la operación del AR.drone, el cual permite capturar las imágenes del patrón de calibración en tiempo real desde cualquiera de las cámaras integradas en el robot. ROS guarda los resultados de la calibración en un archivo que permite un fácil uso y acceso a dichos datos.

2.2.3 FUNCIÓN FINDCHESSBOARDCORNERS

Esta función hace uso de varios filtros que son aplicados a la imagen binarizada, lo que resulta en una imagen que muestra los bordes del tablero de ajedrez, es decir detecta cambios bruscos en intensidad. En este punto surge un inconveniente que tiene que ver con el entorno; los bordes o cambios bruscos de intensidad que fueron capturados en la imagen pero hacen parte del entorno y no del tablero de ajedrez también serán detectados como bordes.

Como solución al problema planteado anteriormente y descartar todos los bordes no deseados, se calcula la 'densidad local de borde' para cada uno de los píxeles. Este valor se obtiene evaluando los píxeles cercanos a aquel que está siendo puesto a prueba, la siguiente expresión muestra la forma exacta de realizar el cálculo:

$$\frac{1}{N}\sum_{i=-50}^{i=50}\sum_{j=-50}^{j=50} BI(x+i, y+j) \quad (10)$$

En la expresión anterior N es el número de píxeles cercanos a tener en cuenta. Como es de imaginarse, este cálculo es bastante complicado computacionalmente hablando, es decir necesita de bastantes recursos para realizarse en poco tiempo, o en su defecto necesita de bastante tiempo para completarse.

Por lo anterior el cálculo se realiza mediante una transformación de imagen llamada integral de imagen, comúnmente usada para tener un gran rendimiento computacional. Este método lo que busca es una suma rápida de un conjunto determinado y cercano de píxeles, este método también es conocido como 'Summed Area Table'.

Para la detección de las esquinas del tablero de ajedrez se tiene en cuenta una serie de ecuaciones que relacionan la geometría del patrón ya conocido con los contrastes observados en los bordes; es decir, de antemano se conoce que una esquina interna estará ubicada justo en el punto en donde se cruzan dos cuadrados oscuros o negros, con dos cuadrados claros o brillantes.

Dejaremos hasta este punto la explicación técnica de la forma en la cual se detectan las esquinas, ya que no es el propósito general de nuestro proyecto, sino es usado como un recurso que será explicado a continuación.

Como se mencionó inicialmente, se harán uso de marcas distribuidas de maneras específicas en el entorno, con el objetivo de conocer la forma en la cual se está moviendo el robot con respecto al mismo. Las marcas específicamente en este caso serán tableros de ajedrez con un tamaño de 4 x 4 cuadros, y es en este punto donde la función findchesscorners de OpenCV es de gran ayuda, ya que permite obtener una ubicación en píxeles de las esquinas internas de cada uno de estos tableros de ajedrez (a los que llamaremos códigos o marcas de aca en adelante).

2.2.4 SOLUCIÓN DEL PROBLEMA DE PERSPECTIVA DE LOS N PUNTOS

El problema de perspectiva de los n puntos es un problema clásico en visión computarizada que se tiene para determinar la pose de una cámara calibrada desde n correspondencias entre puntos de referencia 3D y su proyección en 2D.

Como se menciona en [16], cuando los puntos de control son coplanares (4 puntos), el problema de perspectiva de los 4 puntos (P4P) tiene una única solución.

Para estimar la posición y orientación usando la calibración extrínseca, todos los parámetros intrínsecos de la cámara deben ser conocidos (f: distancia focal, (cx,cy): centros ópticos y $[k_1, k_2, p_1, p_2, k_3]$: parámetros de distorsión). La solución del problema de P4P da la posición (t_{3x1}) y orientación (R_{3x3}: Matriz de rotación) de la cámara con respecto al plano representado por los cuatro puntos (ver figura 3). Se debe asumir que el origen del mundo se encuentra en la esquina superior izquierda del plano rectangular y los cuatro puntos están en el plano z = 0.

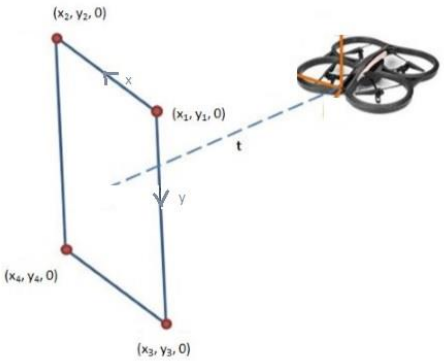


Figura 3. Plano de los cuatro puntos

3. ESPECIFICACIONES

El objetivo final del sistema planteado es calcular y mostrar las coordenadas del AR.drone en tiempo real basándose en un sistema coordenado prestablecido por el usuario, el cual puede ser ubicado a voluntad mediante la localización previa de los códigos en el entorno. Está técnica se conoce en el estado del arte como: *QR code based indoor navigation* [17, 18]

La navegación del AR.drone puede darse de dos formas: la primera de ella consiste en la teleoperación del mismo mediante el paquete ya desarrollado Tum_Ardrone, el cual nos permite controlar el robot con el teclado del computador o un control de PlayStation 3. La segunda forma nace de preestablecer una ruta indicando aceleraciones, ejes y tiempos, mediante el paquete Ardrone_autonomy. Cabe resaltar que la forma en la cual navega el AR.drone no es un factor determinante ya que independiente de la navegación se deben estar obteniendo las coordenadas en tiempo real.

La forma en la cual se efectúa el proceso de localización se describe a continuación:

- ✓ Conexión WiFi de una estación base con el AR.drone.
- ✓ Planeación de la ruta o inicialización del paquete para la teleoperación.
- ✓ Inicialización del nodo slammonocular o slamnodistance garantizando que el robot está observando el primer código.
- ✓ Transmisión de información de la cámara frontal hacia la estación base.
- ✓ Procesamiento de la información en la estación base.
- ✓ Visualización de coordenadas del robot en tiempo real.

Como se puede observar, el proyecto está divido en dos partes fundamentales para su funcionamiento: Primero el AR.drone que será el encargado de transmitir las imágenes y datos a ser procesados, y segundo, la estación base que se encargará del procesamiento en tiempo real de los datos recibidos mediante WiFi que están siendo transmitidos por el AR.drone.

La estación base está conformada básicamente por un computador que debe reunir los siguientes requisitos para garantizar la operación en tiempo real satisfactoriamente:

- ✓ Sistema Operativo: Ubuntu 12.04. De esta manera se asegura que la versión de ROS usada (en este caso ROS fuerte) es completamente compatible, y por lo tanto el nodo a ejecutar no presentará problemas.
- ✓ Procesador: Core i5 o equivalente.
- ✓ Memoria RAM: 4 Gb.
- ✓ En caso de hacer uso de un joystick se recomienda utilizar un control de PlayStation3 por la compatibilidad tanto con el sistema operativo, como con el paquete usado para la teleoperación.
- ✓ Distribución de ROS: Fuerte
- ✓ Paquetes Ardrone_Autonomy[9], Tum_Ardrone[10] y Ardrone_Simulator[11].

Como se mencionó inicialmente, se hizo el desarrollo de un entorno de simulación en el cual el usuario tendrá la opción de validar si todos los paquetes instalados están funcionando de manera adecuada. Para poder usar dicho entorno de simulación es necesario disponer de un computador que tenga una tarjeta de video dedicada, de lo contrario el entorno de simulación tardará demasiado tiempo en cargar y adicionalmente el tiempo de respuesta será demasiado bajo, de tal manera que la experiencia no será satisfactoria.

Durante el desarrollo del proyecto se usaron dos estaciones base diferentes: un computador fijo y uno portátil. En el computador portátil es posible ejecutar los nodos slammonocular y slamnodistance sin

ningún tipo de problemas, sin embargo es imposible hacer uso del entorno de simulación. En el computador fijo se puede hacer uso sin ningún tipo de problemas de los nodos slammonocular y slamnodistance y del entorno de simulación. Las características de cada uno de los computadores son mostradas en la tabla 1.

	Computador Portátil	Computador Fijo
Procesador	Intel Core i5 2540 M	Intel Core i5 4460
Memoria RAM	4 Gb	4 Gb
Tarjeta de Video	Intel HD Graphics 3000	NVidia GTX 750 Ti
Disco Duro	250 Gb	1 Tb

Tabla 1. Especificaciones Estaciones Base de Desarrollo y Pruebas.

En general, la principal limitación está en la cantidad de imágenes que se deben procesar por segundo teniendo en cuenta el tiempo que tarda en procesar cada una de ellas el algoritmo desarrollado. El algoritmo permite cambiar el tiempo que tardan en mostrarse los resultados procesados. Por defecto se ha dejado que cada segundo se puedan visualizar las coordenadas del robot de tal manera que se pueda tomar una decisión de navegación en el instante adecuado para evitar la colisión del AR.drone.

En cuanto al robot de navegación, en este caso el AR.drone 2.0 se puede decir que es un UAV de bajo costo que ofrece unas características propicias para la investigación y avances en temas relacionados con robótica, control y procesamiento de imágenes.

El AR.drone incluye:

- ✓ Procesador ARM Cortex A8 (1 GHz).
- ✓ DSP de video TMS320DMC64x.
- ✓ Cámara frontal HD a 720p – 30 fps.
- ✓ Cámara QVGA vertical 60 fps.
- ✓ IMU (Unidad de medición inercial)

Dado que el AR.drone es un robot de bajo costo, presenta ciertas limitaciones que son determinantes dado que tendrán implicaciones en los resultados reales que se obtienen al ejecutar el algoritmo. La primera limitante con la que nos hemos encontrado es la altura sobre el nivel del mar a la cual el robot puede elevarse sin problemas: el proyecto ha sido desarrollado en su totalidad en Bogotá (Colombia), ciudad que se encuentra ubicada a 2600m sobre el nivel del mar. El AR.drone 2.0 tiene de fábrica unas hélices que permiten que se eleve sin problemas hasta una altura máxima de 2000 m sobre el nivel del mar, por lo cual fue necesario adquirir unas hélices diferentes que permiten que el robot se eleve cuando se encuentra por encima de los 2000 m.

Otra de las grandes desventajas que tiene el AR.drone es la duración de la batería: Teniendo en cuenta que la aplicación es desarrollada para entornos interiores donde no existe una afectación directa de la variación de la batería debido a las condiciones climáticas y meteorológicas, se ha determinado que la duración máxima en vuelo de la batería cargada es de aproximadamente 17 minutos. Sin embargo, una vez que el porcentaje de batería cae por debajo del 30%, la velocidad de transmisión de datos y la estabilidad de la conexión WiFi se ve afectada, lo que provoca resultados indeseados en la ejecución de los algoritmos. Teniendo en cuenta lo mencionado es posible afirmar que el tiempo máximo de ejecución de los algoritmos contando con una batería cargada al 100% inicialmente es de 13 minutos.

Finalmente, en cuanto a desventajas vale la pena mencionar el alcance de la conexión WiFi: teóricamente el alcance máximo de la red generada por el AR.drone es de 50m, sin embargo este alcance se ve limitado por la tarjeta de red inalámbrica de la estación base y los objetos contundentes como paredes, muebles, etc que se encuentren físicamente entre la tarjeta de red y el AR.drone. Por lo tanto es muy difícil definir un alcance máximo de la conexión ya que varía fácilmente dependiendo del entorno y equipos utilizados.

En la figura 4 se muestra el diagrama de bloques de la solución propuesta para el SLAM monocular.

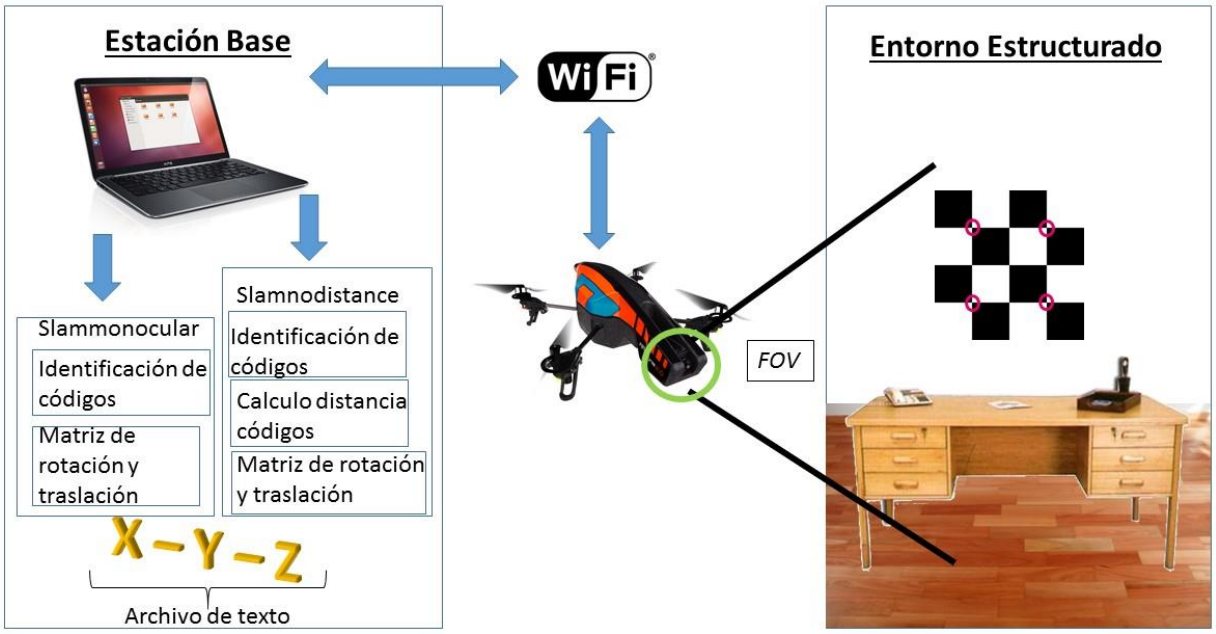


Figura 4. Diagrama de bloques SLAM monocular

El paquete MSlam desarrollado contiene todos los archivos necesarios para una correcta ejecución del algoritmo de localización en tiempo real. Este paquete guarda las coordenadas registradas durante el vuelo en un archivo histórico cuyo nombre puede ser modificado en caso de realizar vuelos con diferentes fines. Este paquete contiene dos nodos de ROS diferentes que ejecutan la localización dadas diferentes condiciones.

El primer nodo llamado "slammonocular.cpp" debe ser usado únicamente cuando se conoce la distancia en 'x' y 'y' entre los diferentes códigos ubicados previamente por el usuario de tal manera que el primer código observado será origen del sistema de referencia y el orden secuencial de códigos debe ser indicado pegando círculos sobre el código; por ejemplo si es el segundo código se deben pegar 1 círculos sobre el código para que el algoritmo se ejecute de manera correcta.

En la tabla 2 se muestran las especificaciones bajo las cuales puede ser usado este primer nodo.

Distancia perpendicular mínima hasta el código	0.6 m
Distancia perpendicular máxima hasta el código	1.2 m
Distancia vertical mínima entre códigos	0.8 m
Distancia horizontal mínima entre códigos	1.3 m
Iluminación	Alta, distribuida de manera uniforme
Distancia máxima entre la estación base y el AR.drone	15 m
Ángulo entre la pared y el robot	$45 < \alpha < 135$
Diámetro de círculos	0.14 m
Ancho del borde de cada uno de los círculos	0.01 m

Tabla 2. Especificaciones para el uso de "slammonocular.cpp".

El segundo nodo llamado “slamnodistance.cpp” debe ser usado cuando la distancia entre códigos no es conocida de manera precisa por el usuario sino que debe ser calculada directamente por el algoritmo. En la tabla 3 se muestran las especificaciones bajo las cuales debe ser ejecutado este segundo nodo

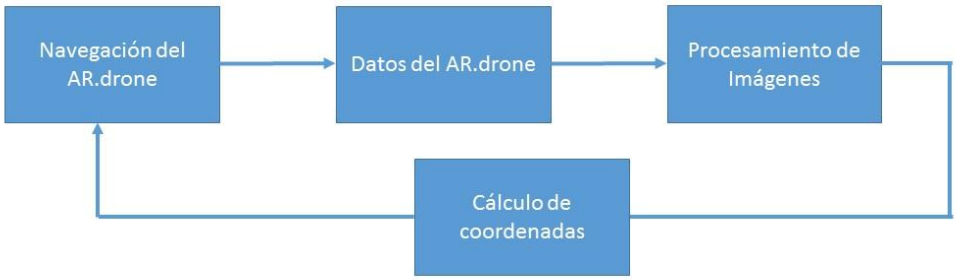
Distancia perpendicular mínima hasta el código	0.9 m
Distancia perpendicular máxima hasta el código	1.4 m
Distancia vertical mínima entre códigos	0.7 m
Distancia horizontal mínima entre códigos	0.7 m
Distancia horizontal máxima entre códigos	1.2 m
Distancia vertical máxima entre códigos	1 m
Iluminación	Alta, distribuida de manera uniforme
Distancia máxima entre la estación base y el AR.drone	15 m
Ángulo entre la pared y el robot	$45 < \alpha < 135$

Tabla 3. Especificaciones para el uso de “slamnodistance.cpp”.

En la figura 5 se muestra la estructura del paquete MSlam.

Figura 5. Estructura del paquete MSlam.

Cada uno de los algoritmos tienen un lazo de control que se encarga de realimentar la información que está siendo suministrada por la cámara frontal del AR.drone a la persona que está en el comando central, es decir en la estación base, ya que con la información visualizada en pantalla puede tomar alguna acción de control sobre el vuelo del robot. En el cuadro 1 se muestra el lazo de control generalizado mediante el cual operan los algoritmos desarrollados.



Cuadro 1. Lazo de control de los algoritmos desarrollados

4. DESARROLLOS

Este proyecto se basa en el desarrollo de algoritmos de visión con el fin de obtener la ubicación del AR.drone en un entorno desconocido para él (SLAM), a partir de códigos ubicados en el entorno, con respecto a un sistema de coordenadas (x,y,z) con su origen en un punto determinado del primer código observado, para esto se usó ROS y otras librerías que facilitan el desarrollo de dichos algoritmos.

Para llevar a cabo estos algoritmos se debe hacer uso de paquetes de ROS que son unidades para la organización del software. El paquete "MSlam" desarrollado en este proyecto implica la creación de otros archivos para su funcionamiento como lo son:

- ✓ Manifest: En este archivo se deben especificar las dependencias que tiene el paquete de librerías u otros paquetes de ROS, junto con los datos de los autores del paquete, en este archivo también se puede especificar el tipo de licencia usada para el paquete. Para nuestro proyecto algunas de las dependencias utilizadas son:
    - roscpp: Esta dependencia contiene el compilador de C++ en ROS y es necesario para la compilación del código, ya que todo fue desarrollado en lenguaje de programación C++.
    - OpenCV2: Librería para el procesamiento de las imágenes que se van a estar recibiendo de la cámara frontal del AR.drone.
    - cv_bridge: Este paquete se encuentra contenido en ROS y se usa para la transformación de imágenes de ROS a OpenCV o viceversa, es necesario para poder realizar modificaciones a la imagen recibida desde OpenCV y poder hacer el análisis de las mismas ya que con la imagen en ROS no se podría llevar a cabo todo este procesamiento.
    - ardrone_autonomy: Paquete utilizado para la planeación de la trayectoria que debe realizar el AR.drone dentro del entorno y que genera la secuencia de video en tiempo real a la cual se debe hacer una suscripción para el procesamiento.
- ✓ Mainpage: En este archivo se debe escribir la documentación del paquete, como lo son los comentarios acerca de los ejecutables del paquete.
- ✓ CMakelists: Este archivo hace la construcción del paquete, es decir, especifica cómo construir el paquete, incluye el macro *rosbuild_init()* que define la ubicación de salida de los ejecutables y librerías, también incluye el path o la dirección donde se ejecutara el paquete e invoca rospack para la compilación de este. En este archivo también se debe especificar los algoritmos que serán definidos como nodos mediante la orden *rosbuild_add_executable()* para su ejecución en ROS.
- ✓ Makefile: Este archivo contiene la instrucción cmake para la ejecución del archivo CMakelists que es fundamental para el funcionamiento de todos los nodos que contenga el paquete.

Luego de construir el paquete Mslam, se realizó el desarrollo de los archivos fuentes o nodos del paquete por lo que en las siguientes secciones se explicará el desarrollo de cada uno de estos nodos, el procedimiento seguido para la calibración de la cámara y la creación del entorno de simulación.

4.1 PROCEDIMIENTO DE CALIBRACIÓN CÁMARA FRONTAL AR.DRONE

Como se ha mencionado anteriormente, en el desarrollo de este proyecto se ha hecho uso de paquetes y nodos ya existentes que permiten la teleoperación del robot y el acceso a datos de sensores, navegación, video en vivo de las cámaras disponibles en el AR.drone, etc.

Uno de los nodos usados para el proyecto fue el llamado "camera_calibrator". Este nodo permite encontrar los valores de los parámetros intrínsecos para una cámara. Para el uso de este nodo es necesario tener un tópico en el cual se pueda acceder al video en vivo emitido por la cámara que se requiere calibrar.

En este caso, el tópico en el cual se publica el video en vivo de la cámara frontal es "/ardrone/front/image_raw", el paquete encargado de la publicación de este tópico es ardrone_autonomy, el cual es un paquete ya desarrollado y usado específicamente para acceder a datos de navegación del AR.drone

Para la calibración de la cámara basta con hacer el llamado al nodo camera_calibrator, el cual se encarga de realizar la calibración utilizando como patrones tableros de ajedrez. Dentro del llamado es necesario indicar el tamaño del tablero de ajedrez y el tamaño real en metros de cada uno de sus cuadros. Luego de hacer el llamado, se abre de manera automática una interfaz gráfica donde se puede observar el video en vivo de la cámara que se quiere calibrar.

Para que el proceso de calibración sea efectivo, es necesario mover el tablero de ajedrez de tal manera que se tengan capturas desde diferentes puntos de vista del patrón de calibración, variando la distancia y el ángulo desde el cual se captura la imagen de dicho patrón. Para este caso se tuvo un total de 40 posiciones diferentes sobre las cuales se realizó el proceso de calibración obteniendo los siguientes resultados:

$$Matriz\ de\ calibración = \begin{bmatrix} 576.570022776563 & 0 & 300.046985405385 \\ 0 & 561.678422453689 & 186.260649273004 \\ 0 & 0 & 1 \end{bmatrix} \quad (11)$$

$$Parámetros\ de\ Distorsión = [-0.5149194 \quad 0.257843950 \quad -0.00388223 \quad 0.00345349 \quad 0] \quad (12)$$

La forma en la cual se encuentran organizadas las matrices anteriores está descrita en la sección 2.2.2.

4.2 ALGORITMO SLAMMONOCULAR.CPP

El objetivo principal del algoritmo de slammonocular.cpp es dar la ubicación del AR.drone con respecto al eje de coordenadas a medida que realiza una trayectoria observando uno a uno los tableros de ajedrez en un orden determinado, conociendo la distancia entre estos a priori, es decir, los tableros de ajedrez estarán distribuidos a lo largo de una pared de forma tal que se conoce la distancia entre cada uno de ellos (en el eje 'x' y 'y'), esta distancia es utilizada en el código fuente para determinar la ubicación del AR.drone.

Como todo el algoritmo se basa en las imágenes que se están recibiendo constantemente de la cámara frontal lo primero que se debe asegurar es poder obtener las imágenes de la cámara y pasarlas a un formato entendible para su procesamiento en OpenCV, esto se logra haciendo uso de la dependencia de cv_bridge, este proceso se describe en el diagrama de flujo mostrado en la figura 6.

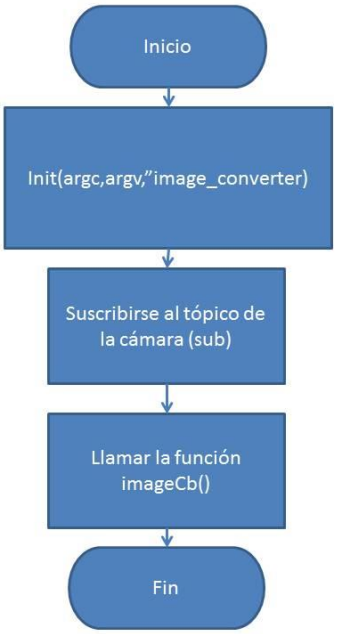


Figura 6. Diagrama de flujo para recibir la imagen

Este programa comienza con la activación de la función *init()* de ROS, la cual se encarga de la inicialización del código y ejecución del nodo. Luego se realiza la creación de la variable "sub" (variable de tipo suscriptor), la cual se suscribe al tópico /ardrone/front/image_raw y le envía ese dato a la función *imageCb()*, en este punto cabe recalcar que al suscribirse a un tópico en ROS se crea un ciclo infinito, donde constantemente se están recibiendo los datos publicados en este tópico mientras estos estén disponibles, es decir, en nuestro caso se estará recibiendo constantemente la imagen de la cámara frontal y enviándola a la función *imageCb()* mientras la conexión WiFi sea estable.

Cuando se ingresa a la función *imageCb()* se realiza la transformación de la imagen del formato sensor_msgs, que es el formato en el que ROS entrega la imagen, a un formato BGR8, que es un formato para las imágenes a color, con el objetivo de poder leerla y utilizarla con las funciones de OpenCV.Luego de realizar esta transformación, la imagen es almacenada en un apuntador global, esto se realiza para poder tener disponible la imagen a lo largo del todo el programa y no solo cuando se encuentre dentro de esta función.

Cuando *imageCb()* no recibe ningún mensaje de ROS, mostrará un mensaje en pantalla de error donde se avisa al usuario que la cámara del AR.drone no está enviando información para ser procesada.

Luego de este proceso se debe hacer una rectificación de la imagen recibida, esto por lo que se explicó en la calibración de la cámara en el marco teórico. La imagen que se recibe directamente de la cámara viene con distorsiones como se muestra en la figura 7, donde se pueden observar las curvaturas que deben eliminarse para el proceso de SLAM, dado que para los cálculos de profundidades de la imagen se presenta un alto nivel de error al no tener una imagen plana o rectificada.

Figura 7. Imagen cámara frontal AR.drone sin rectificar.

Es por esto que se implementó la función *rectificarimagen()* como se muestra en el siguiente diagrama de flujo(ver figura 8).

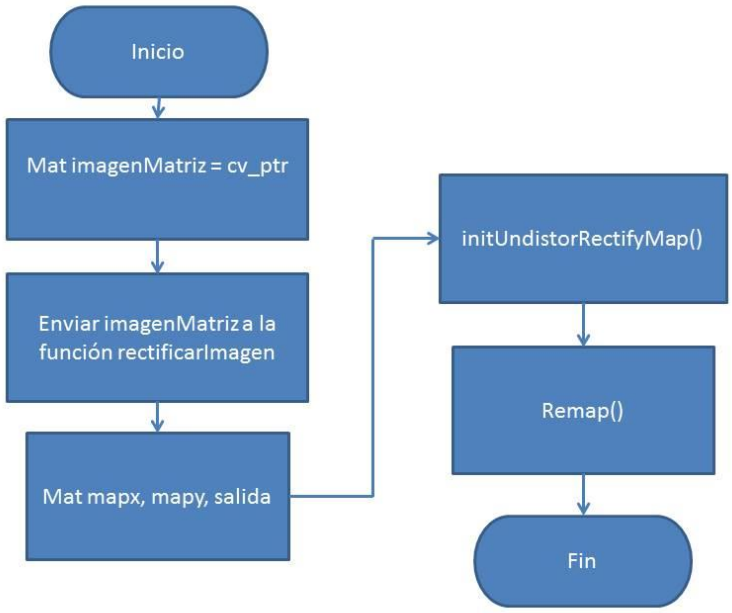


Figura 8. Diagrama de flujo para rectificar imagen

Lo primero que hace este código es pasar lo que tiene el apuntador global "cv_ptr" antes mencionado, a una estructura de tipo MAT que es un tipo de estructura para OpenCV, el cual representa una imagen y contiene ciertos valores para su caracterización como por ejemplo el tamaño de la imagen, en que formato se tiene, entre otras. Luego esta variable se pasa a la función de *rectificarImagen()* y lo que retorna esta función también es almacenado en otra estructura de tipo MAT.

Cuando se ingresa a la función *rectificarImagen()* lo primero que se hace es declarar tres estructuras de tipo MAT que se llenarán dentro de esta función.

Luego se hace uso de funciones ya implementadas en OpenCV como lo es *initUndistortRectifyMap()*, la cual debe recibir como parámetros de entrada los parámetros intrínsecos obtenidos en la calibración de la

cámara y los coeficientes de distorsión que también son obtenidos en el proceso de calibración. Otro de los parámetros de entrada es el tamaño de la imagen a rectificar que como se explicó antes es uno de los valores ya almacenados en la estructura tipo MAT, por lo que basta con hacer un llamado a la función (que hace las veces de método) *.size()* y se retornará dicho valor de la estructura que tiene almacenada nuestra imagen.

El objetivo de la función *initUndistortRectifyMap()* es calcular la transformación y rectificación de la distorsión que finalmente es representada con mapas de reasignación, que son utilizados en la siguiente función del seudocódigo llamada *remap()*, que es una función también implementada en OpenCV donde partir de los mapas de reasignación calculará los nuevas ubicaciones en pixeles de la imagen corregida y las asignará a una matriz de salida, es decir, que con el resultado de la función *initUndistortRectifyMap()*, se construye una nueva imagen haciendo la reubicación de cada uno de los pixeles en una nueva variable tipo MAT que es el resultado final de nuestra función *rectificarimagen()* y serán estas imágenes rectificadas las que se usen para hacer todos los cálculos correspondientes.

Para lograr hacer una comparación entre la imagen recibida (ver figura 7) y la imagen rectificada (ver figura 9), se ha tomado una fotografía donde se puede ver con facilidad el efecto de la rectificación. Es apreciable el cambio en la concavidad de la imagen sin rectificar y la imagen rectificada que se ve totalmente plana aunque cabe aclarar que en la figura 9 en donde está marcado con amarillo se aprecia el borde torcido pero esto no es síntoma de que la rectificación este mal si no que ese borde en realidad se encuentra de esa manera.

Figura 9. Imagen cámara frontal AR.drone rectificada.

Luego de este proceso de rectificación se debe realizar un nuevo proceso antes de iniciar el análisis de la imagen, este proceso es la conversión de la imagen que se está recibiendo a color (RGB) a una imagen en escala de grises, este proceso es necesario para un mejor rendimiento y un proceso más rápido en el análisis de las imágenes.

Una mejor explicación por la cual este paso debe realizarse es que una imagen en escala de grises está compuesta por píxeles representados por múltiples bits de información en un único canal de 8 bits, en cambio una imagen a color está representada por una profundidad de bits entre 8 y 32. Entonces para representar los colores se utilizan combinaciones de esos bits, consiguiendo un total de colores mucho más amplio que si se trabaja solo en escala de grises, haciendo que la información de la imagen sea muy pesada y compleja de procesar por el problema que representa tratar con tantos bits al tiempo.

Es por esto que también se hace uso de la función *cvtColor()* de OpenCV la cual hace este proceso de conversión de una imagen RGB a escala de grises, perdiendo así información sobre los colores pero no información sobre tamaño de objetos específicos en la imagen, es decir en general la única información que se pierde es el color que para esta aplicación no es un factor importante o determinante.

Ya con estos cambios aplicados a cada una de las imágenes que se van recibiendo del AR.drone se puede realizar el proceso de SLAM conociendo la distancia entre los códigos, en la figura 10 se muestra un diagrama de flujo de cómo se realizó este proceso.

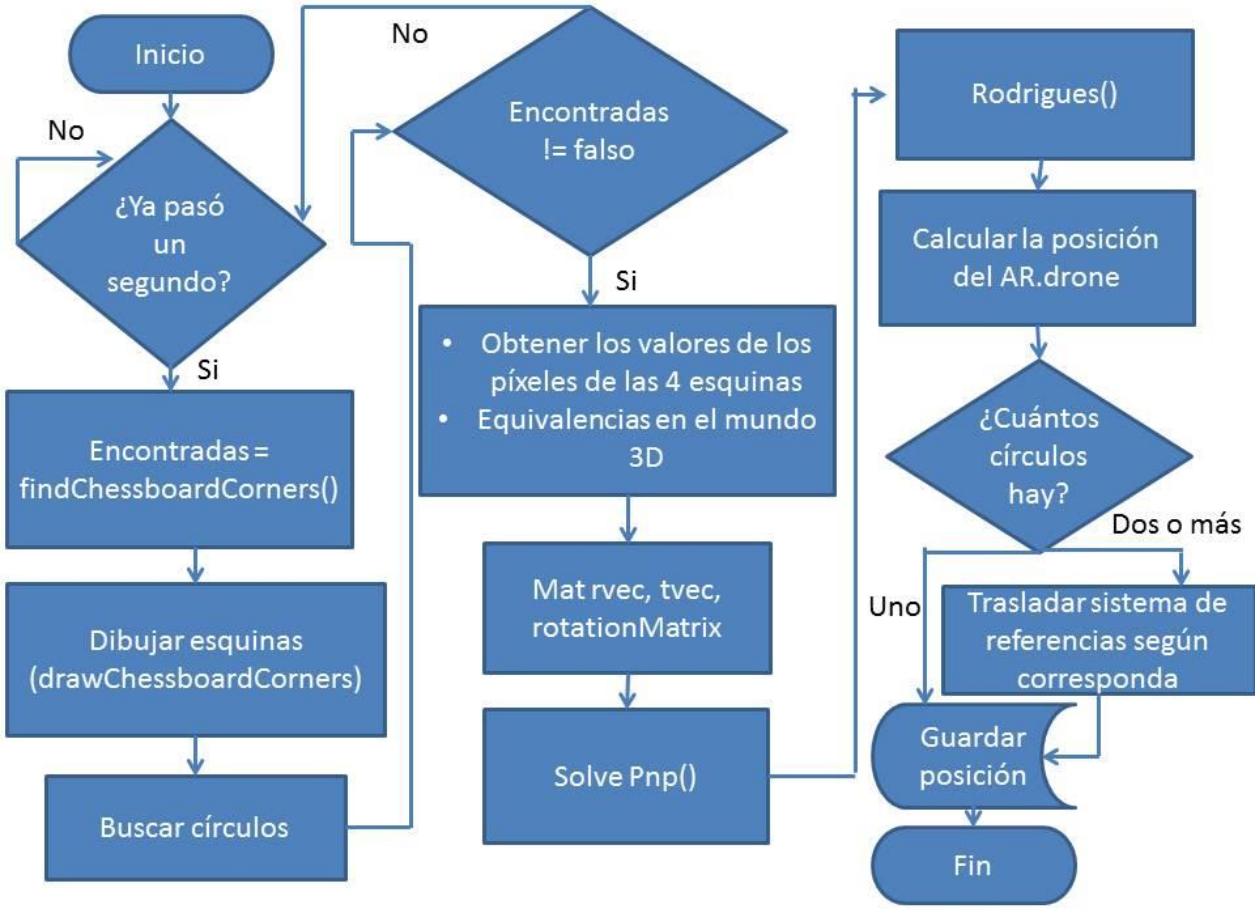


Figura 10. Proceso de SLAM conociendo distancias

En la primera parte del código es apreciable que se hace una pregunta sobre el tiempo que ha transcurrido, esta pregunta se hace porque al hacer la suscripción al tópico de la cámara se están recibiendo imágenes a 30 frames por segundo, es decir que en cada segundo se están recibiendo 30 imágenes, lo cual es una frecuencia demasiado alta para esta aplicación, ya que no se espera que en 33 ms el AR.drone haya cambiado tanto su posición como para tener que calcularla con esta frecuencia, es por esto que se realiza ese *if()* sobre si ya paso un segundo para solo hacer el cálculo de la posición del AR.drone cada segundo, donde sí se pudo tener un desplazamiento con respecto al anterior segundo.

Ahora, luego de que ha pasado un segundo, se debe hacer la búsqueda de las esquinas del código en la imagen que se tiene en este momento que como se explicó en la sección 2.2.3 se hará mediante el uso de la función *findChessboardCorners()* la cual devuelve una variable de tipo booleano con un valor de "verdadero" si ha sido posible detectar las esquinas de un código en la imagen que se le ingresa o un valor de "falso" en caso contrario.

A la función *findChessboardCorners()* se le debe ingresar como parámetros de entrada la imagen en la cual se debe buscar el código, el tamaño del código, es decir cuántos cuadros tiene a lo ancho y a lo largo y un vector el cual será llenado por la función con los valores de los pixeles donde se encuentran las esquinas de los cuadros del tablero de ajedrez en caso de que estos sean encontrados, cuando no son encontrados este vector se mantiene con los valores que se le hayan ingresado a la función.

Luego de haber encontrado las esquinas en dicha imagen se usa la función *drawChessboardCorners()* que permitirá dibujar en la imagen que se está analizando las esquinas que se encontraron, esto simplemente se usa para mostrar al usuario que las esquinas de los códigos se están encontrando correctamente y corresponden a las esquinas interiores del código y no a otro tipo de esquinas que se puedan estar observando en la misma imagen, es por esto que esta imagen usa el vector que se llena en la anterior función para dibujar círculos alrededor de los píxeles que fueron marcados como esquinas.

Como todos los códigos distribuidos en la pared son iguales, se debe tener alguna distinción en el código que permita saber con exactitud cuál de ellos está siendo observado, es por esto que se recurrió a agregar a cada código un número de círculos dependiendo de qué código sea; es decir, el primer código que se observe y en el cual estará fijado el sistema de referencia del mundo como se muestra en la figura 11 no se tendrá ningún circulo, pero en el siguiente código que se espera se visualice deberá tener un circulo, el siguiente dos círculos y así sucesivamente para saber por qué código se ha pasado y cuál es, lo que permite saber a qué distancia se encuentra este código del primero observado.

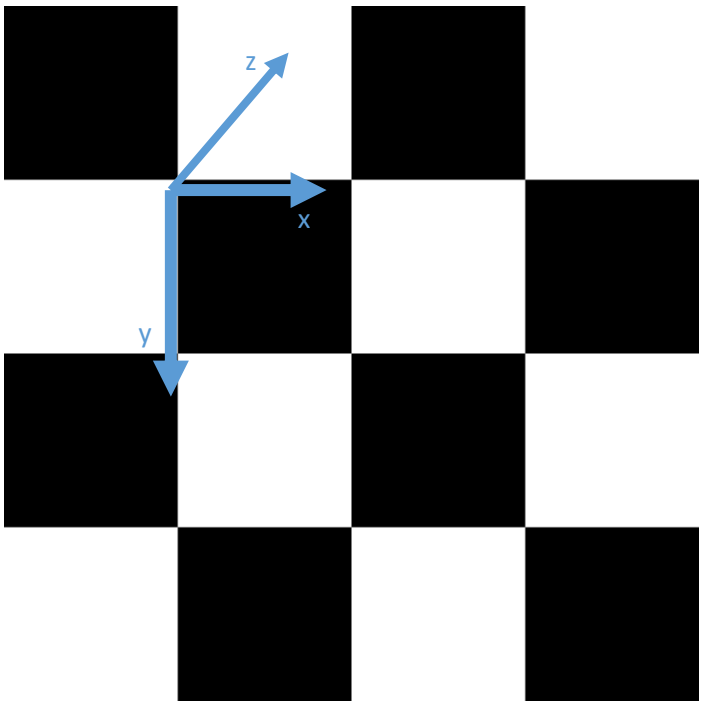


Figura 11. Sistema de referencia sobre los códigos

Por lo anteriormente mencionado se ve la necesidad de usar la función de OpenCV *HoughCircles()* que es una función que permite encontrar círculos en una imagen en escala de grises, está función debe recibir como parámetros la imagen en la cual se deben encontrar los círculos, un vector el cual se llenará en esta función con la ubicación en pixeles del centro del circulo y el radio del mismo, el método que debe usar la función para encontrar los círculos pero que actualmente solo permite un método llamado transformada de Hough que es una técnica para la detección de figuras en imágenes que primero detecta bordes en la imagen para obtener los puntos de la imagen que pertenecen a la frontera de la figura deseada y como se

espera que en la imagen no se pueda observar todo un borde constante la transformada de Hough hace un análisis de cómo puede estar distribuido este borde y así determina si pertenece a la figura que se busca.

Luego de este proceso de buscar las esquinas de los códigos y los círculos en la imagen se debe preguntar si realmente se encontraron esquinas que es la labor del *if(encontradas != falso)* que lo único que hace es revisar el booleano devuelto por la función *findChessboardCorners()* y si este es diferente de falso se hará el proceso para el cálculo de la posición si no se debe esperar a una imagen donde si haya un código.

Cuando este booleano toma un valor de verdadero es el momento de determinar la posición del AR.drone con respecto al sistema de referencia del mundo, con los valores devueltos por las funciones anteriormente mencionadas.

Como se explicó en la sección 2.2.4 para tener una única solución al problema de perspectiva de los n puntos se deben tener 4 puntos coplanares, es por esto que primero se deben seleccionar cuales van a ser estos cuatro puntos coplanares a usar en la solución de perspectiva, en nuestro caso se seleccionaron los 4 puntos extremos del código como muestra la figura 12, y por esto se debe recuperar la información del vector llenado por la función *findChessboardCorners()* (la cual retorna un valor booleano true en caso de haber encontrado los puntos) sobre la ubicación en píxeles de estas 4 esquinas que son las que se desean.

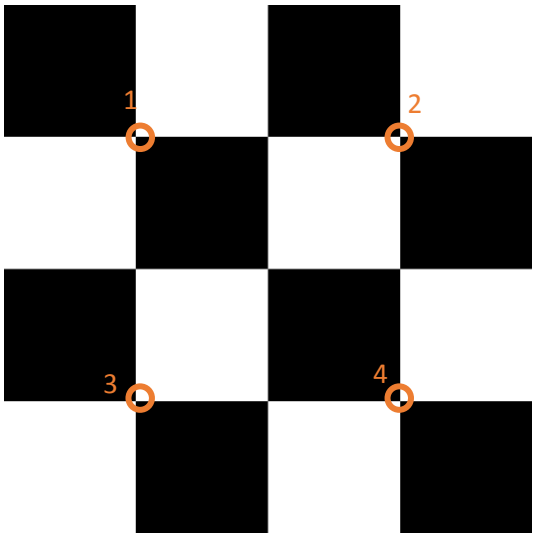


Figura 12. Los 4 puntos que se seleccionan de cada código.

Ahora se deben crear los vectores con las equivalencias de estos 4 puntos en 3D, estos valores están dados en metros a partir del sistema de referencia del mundo (ver figura 11), es decir que todos los puntos en 3D tienen una componente en ‘z’ de 0 y sus componentes en ‘x’ y ‘y’ estarán dadas en metros desde el origen del sistema de referencias.

Luego se crean tres estructuras de tipo MAT que serán llenadas posteriormente, estas tres estructuras se llaman rvec, tvec y rotationMatrix donde se almacenarán los valores de los vectores de rotación, traslación y matriz de rotación, respectivamente.

El vector de rotación es la misma matriz de rotación sino que representada en forma de un vector fila de tres posiciones, este vector contiene información sobre la forma como están rotados los ejes en la cámara con respecto al mundo pero la matriz de rotación contiene información de cómo están rotados estos ejes con respecto a todos los ángulos en los que se puede mover el AR.drone y es por esto que se usa la matriz de rotación y no el vector de rotación.

Las estructuras rvec y tvec son los valores que llenará la función *solvePnP()*, la cual recibe como parámetros de entrada los valores en píxeles de las esquinas del código, los valores de estas esquinas en el mundo 3D, la matriz con los parámetros intrínsecos de la cámara y la matriz de distorsión, esta matriz de distorsión en este momento se le pasa con todos los valores en cero por lo que la imagen se encuentra rectificada y se supone que ya no tiene ninguna distorsión. Con todos estos parámetros de entrada esta función encuentra el vector de traslación y de rotación haciendo un cálculo de la equivalencia de los puntos en 3D y los de la imagen teniendo en cuenta la característica de la cámara.

Como se mencionó anteriormente se tiene más información acerca de la rotación con la matriz que con el vector y es por esto que se hace un llamado a la función *Rodrigues()* la cual usa las expresión 13 a 15 para obtener la matriz de rotación. Esta función también se encuentra implementada en OpenCV.

$$\theta = norm(r) \qquad (13)$$

$$r \leftarrow \frac{r}{\theta} \qquad (14)$$

$$R = \cos(\theta * I) + (1 - \cos\theta) * r * r^T + \sin\theta * \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} \qquad (15)$$

(16) $$Matriz\ de\ rotación\ (R) = \begin{pmatrix} c\theta c\psi & c\theta s\psi & -s\theta \\ s\phi s\theta c\psi - c\phi s\psi & s\phi s\theta s\psi + c\phi c\psi & s\phi c\theta \\ c\phi s\theta c\psi + s\phi s\psi & c\phi s\theta s\psi - s\phi c\psi & c\phi c\theta \end{pmatrix}$$

Donde θ es el ángulo de pitch, ϕ es el ángulo de roll y ψ es yaw.

Como todos estos valores de rotación y de traslación calculados están con respecto a la cámara y lo que se desea saber es la ubicación de la cámara con respecto al mundo se debe hacer los cálculos correspondientes que están dados por la siguiente ecuación:

$$Posición\ en\ el\ mundo$$

$$(x, y, z) = -(Matriz\ de\ rotación\ de\ la\ cámara)^{-1} * (vector\ de\ traslación) \quad (17)$$

Es por este fin que primero se calcula la inversa de la matriz de rotación obtenida a partir de la función *Rodrigues()* y luego se hacen las operaciones correspondientes; la multiplicación entre esta nueva matriz y el vector de traslación obteniendo así un vector fila de tres posiciones, el cual contiene la distancia en metros a la que se encuentra el centro de la cámara del AR.drone en 'x', 'y' y 'z' con respecto al sistema de referencias.

Estos valores que se obtienen hasta este momento son con respecto al sistema de coordenadas ubicado en cada código observado y es por esto que toma importancia el saber qué código se está observando, y es por esto que se hacen los siguientes *if()* acerca del número de círculos porque si no se está viendo ningún circulo se sabe que estos valores son correctos ya que están referenciados al origen del mundo pero ya si se observa algún circulo estos valores no estarán referenciados con este origen.

En esta rutina, como es conocida la distancia entre cada uno de los códigos, lo que se hace dentro de cada uno de estos *if()* es desplazar ese resultado en 'x' y 'y' a la posición del origen del mundo, es decir que por ejemplo el segundo código donde se encuentra solo un circulo está a una distancia del primer código en 'x' de 0.1 m y en 'y' de -1 m, se le realizará este corrimiento a lo que se obtuvo con respecto a este código y así sucesivamente para los códigos que se vayan observando.

Se debe resaltar que estas distancias entre los códigos deben ser coherentes con respecto al sentido de los ejes de coordenadas, es decir que si el segundo código observado se encuentra por debajo del primero el desplazamiento en 'y' es positivo pero si se encuentra por encima debe ser negativo y para los casos de 'x' si el código se encuentra a la derecha del que será el primer código debe tener un desplazamiento positivo pero si se encuentra a la izquierda este desplazamiento es negativo, también es importante aclarar que el eje 'z' no sufre ningún desplazamiento ya que se supone que todos los códigos se encuentran sobre una misma pared que no sufre ningún cambio de profundidad a medida que se hace el desplazamiento del AR.drone.

En el caso de que esta suposición no se pueda cumplir se debe modificar la rutina para que tenga en cuenta también estos desplazamientos en 'z' que se haría de la misma manera que se hace con 'x' y 'y' con distancias conocidas y medidas antes de ejecutar la rutina.

Finalmente se vuelve a poner el booleano que nos indica si fueron encontradas esquinas en la imagen en falso, esto para que en la siguiente imagen a analizar no se tenga un valor que no venga de la función *findChessboardCorners()* y se le muestra al usuario en una ventana que se abrirá en la pantalla del computador la imagen que se está analizando en ese momento que como se mencionó antes, si se encuentra el código también mostrará las esquinas marcadas del código y las que se están usando para todo el proceso ya mencionado.

En la figura 13 se muestra el diagrama de flujo que resume el proceso que ejecuta el algoritmo.

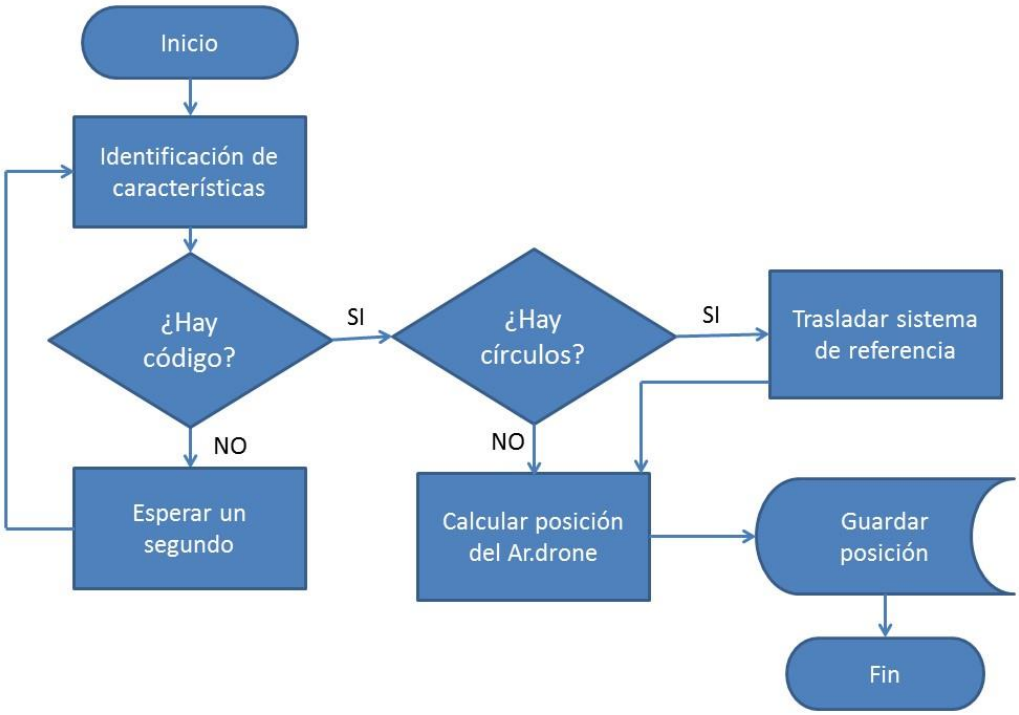


Figura 13. Diagrama de flujo algoritmo slammonocular

4.3 ALGORITMO SLAMNODISTANCE.CPP

El objetivo principal del algoritmo de slamnodistance.cpp es dar la ubicación del AR.drone con respecto al eje de coordenadas a medida que realiza una trayectoria observando uno a uno los tableros de ajedrez en

un orden determinado, sin conocer la distancia de separación entre cada uno de estos códigos, es decir, los tableros de ajedrez estarán distribuidos a lo largo de una pared de forma tal que no se conoce la distancia entre cada uno de ellos (en el eje 'x' y 'y'), y el algoritmo debe ser capaz de calcular la distancia entre cada uno de estos códigos para así lograr dar una ubicación del AR.drone con respecto al sistema de coordenadas a medida que hace su recorrido.

En este algoritmo al igual que en el anterior se implementaron los primeros diagramas de flujo (ver figura 6 y 8), explicados anteriormente para recibir las imágenes, llevar a cabo la rectificación de las mismas y su paso a escala de grises.

En este algoritmo se vio la necesidad de crear una función llamada *calcularDistancia()*, la cual se encarga de calcular las distancias en 'x', 'y' y 'z' del AR.drone con respecto al sistema de coordenadas (ver figura 11) de cada uno de los códigos, el diagrama de flujo de esta función se muestra en la figura 14.

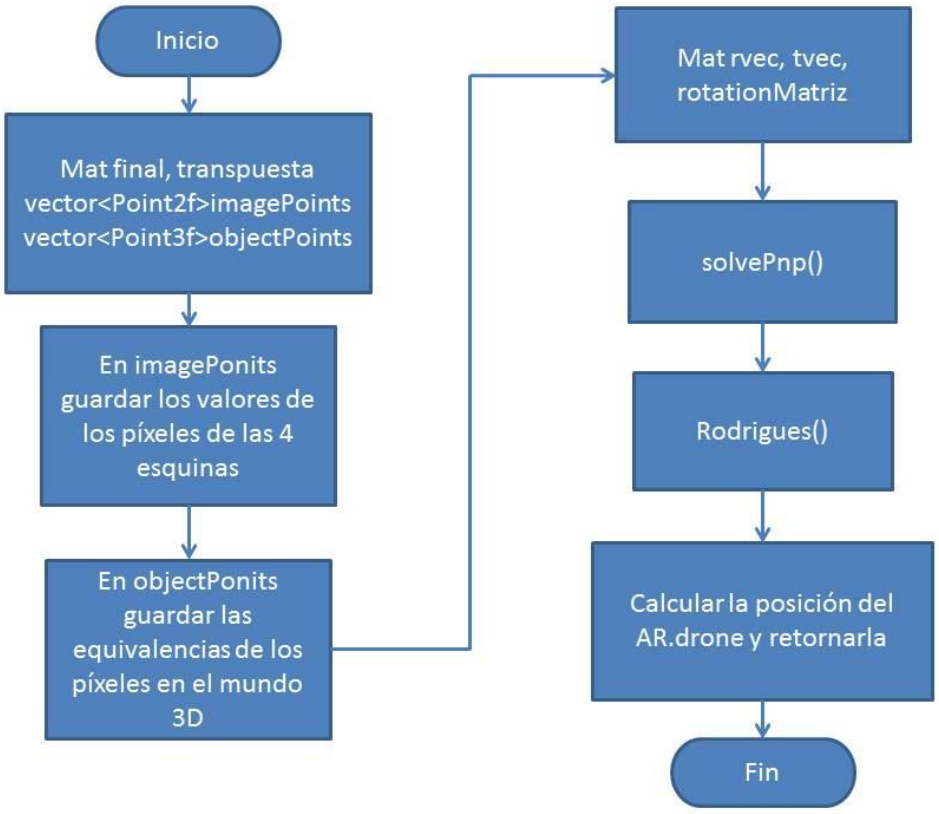


Figura 14. Diagrama de flujo de la función calcularDistancia()

Esta función recibe un parámetro de tipo *vector<Point2f>* llamado imageCorners, este tipo de variable para OpenCV corresponde a un arreglo de vectores que tienen dos coordenadas en este caso 'x' y 'y' que son las posiciones 'x' y 'y' en píxeles de cada una de las esquinas interiores del código, esta función a partir de este parámetro de entrada debe retornar una variable de tipo MAT que corresponderá a la posición del AR.drone en 'x', 'y' y 'z' con respecto al código.

Ahora dentro de la función en los primeros pasos se declaran variables dos de tipo MAT como lo son final y transpuesta, además de esto se declaran otros dos vectores uno de tipo Point2f y otro de Point3f como lo son imagePoints y objectPoints que son las variables donde se almacenaran los valores de los píxeles en 'x' y 'y' donde se encuentran las 4 esquinas del código que se usan para el proceso, y el vector de 3

dimensiones donde se guardan los valores ‘x’, ‘y’ y ‘z’ del código en metros al igual que se hacía en el código anterior.

Los demás pasos corresponden a lo mismo que se realizaba en la figura 10 con la solución al problema de perspectiva de los 4 puntos y el cálculo de la posición del AR.drone con las matrices de rotación y de traslación obtenidas a partir de la función *solvePnP*.

Luego de implementar esta función se implementó otra función llamada *distanciaCodigo()*, que se muestra en la figura 15, la cual será la encargada de calcular de distancia en ‘x’ y ‘y’ entre cada uno de los códigos a partir de una imagen donde se puedan observar dos códigos.

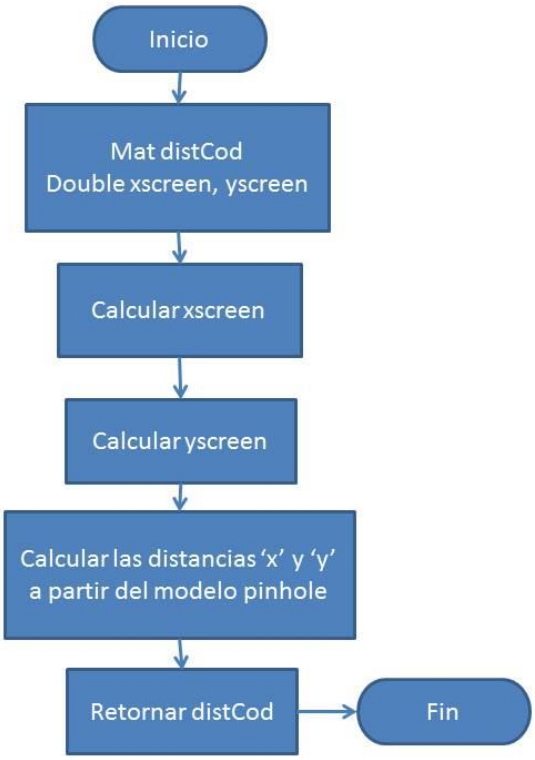


Figura 15. Diagrama de flujo de la función distanciaCodigo()

Esta función recibe 6 parámetros de entrada de tipo double llamados imagen1x, imagen1y, imagen2x, imagen2y y promedioZ los cuales corresponden al valor del píxel en ‘x’ y ‘y’ del origen en cada uno de los códigos que se están observando en esa imagen y el valor ‘z’ a la que se encuentran los códigos del AR.drone.

Dentro de la función primero se declara la variable tipo MAT que retornará la función llamada distCod y que contendrá la distancia en metros que hay entre los dos códigos en ‘x’ y en ‘y’, también se declaran dos variables de tipo double “xscreen” y “yscreen” que son las variables que se llenan a continuación con la distancia entre los códigos pero en píxeles.

En este punto cabe aclarar que para conocer la distancia entre dos códigos se debe descomponer la imagen original en dos una parte derecha y otra parte izquierda para que en cada una de estas partes se pueda encontrar un código, ya que de no hacerse con dos imágenes así estén presentes los códigos cuando se buscan esquinas solo se hayan las de un código, es por esto que en toda la implementación del algoritmo se divide cada imagen en dos y se busca en las dos partes códigos.

A raíz de lo aclarado anteriormente para calcular la distancia en píxeles entres los códigos en 'x' se debe usar la siguiente ecuación:

$$xscreen = |pixel\ en\ x\ código\ izquierda - (320 + pixel\ en\ x\ código\ derecha)| \quad (18)$$

En cambio para calcular la distancia en 'y' se usa

$$yscreen = |pixel\ en\ y\ código\ izquierda - pixel\ en\ y\ código\ derecha)| \quad (19)$$

La diferencia que se encuentra entre las dos ecuaciones es que en el valor del píxel donde se encuentra el origen del código observado a la derecha de la imagen se le debe sumar 320 píxeles por lo que es el valor en el que se corta la imagen original.

Luego de tener estas distancias en píxeles y a partir de las ecuaciones dadas por el modelo pinhole de una cámara pueden calcularse las distancias en metros de 'x' y 'y' como muestran las siguientes ecuaciones:

$$Distancia\ en\ X = Z * \frac{xscreen - cx}{fx} \quad (20)$$

$$Distancia\ en\ Y = Z * \frac{yscreen - cy}{fy} \quad (21)$$

En las dos ecuaciones Z corresponde a la distancia en metros que se encuentra el AR.drone y que se recibe como parámetro a la función y se calcula a partir de la función *calcularDistancia(),* después de realizar este cálculo se almacenan estos valores de las distancias en la matriz disCod para ser retornadas a la función principal.

Para explicar la función principal del este algoritmo se usará el diagrama de flujo de la figura 16, donde se hace el análisis de cada una de las imágenes recibidas.

En la función principal lo que hace primero es crear una imagen que contiene la copia de la imagen que se recibe y se tiene rectificada, luego de esto se hace la división de esta imagen en dos partes que corresponde a la imagen1 donde se almacena la copia de la imagen desde la coordenada (0,0) hasta la (320,360) que sería la mitad de la imagen y en imagen2 se almacena lo correspondiente a la otra mitad de la imagen.

Luego de esto y al igual que en el algoritmo anterior se analiza una imagen cada segundo ya que con este algoritmo tampoco se espera que el AR.drone pueda cambiar de posición drásticamente en un tiempo menor a este y por esto no se analizan las 30 imágenes que se reciben en un segundo de la cámara y es para esto que se usa el primer *if().*

Dentro de este *if()* se buscan esquinas con la función *finChessboardCorners()* en las tres imágenes que se tienen que son la imagen completa y las dos mitades, lo que retorna esta función para cada una de estas imágenes se almacena en variables de tipo bool que para la mitad izquierda se almacena en una variable con el nombre found, para la mitad derecha se almacena en la variable found2 y finalmente en found3 se almacena el resultado para la imagen completa.

A continuación, se tiene otro *if()* donde se pregunta si se encontró esquinas solamente en la mitad izquierda, que correspondería a tener found en verdadero y found1 en falso, en caso de que esto sea afirmativo se considera que en la imagen solo se encuentra un código por lo que se crea una variable de tipo MAT llamada final donde se almacenará la posición en 'x', 'y' y 'z' del AR.drone con respecto a dicho código y esto se hace haciendo un llamado a la función *calcularDistancia(),* explicada anteriormente y a la cual se le pasa como parámetro las posiciones 'x' y 'y' en píxeles de todas las esquinas que encontró en la imagen rectificada, es importante aclarar que se le deben pasar estas esquinas

y no las halladas en la búsqueda correspondiente a la mitad izquierda porque la función *solvePnP()* que se usa dentro de la función *calcularDistancia()* usa los parámetros intrínsecos de la cámara y etsos corresponden a la imagen completa no a la mitad de la imagen.

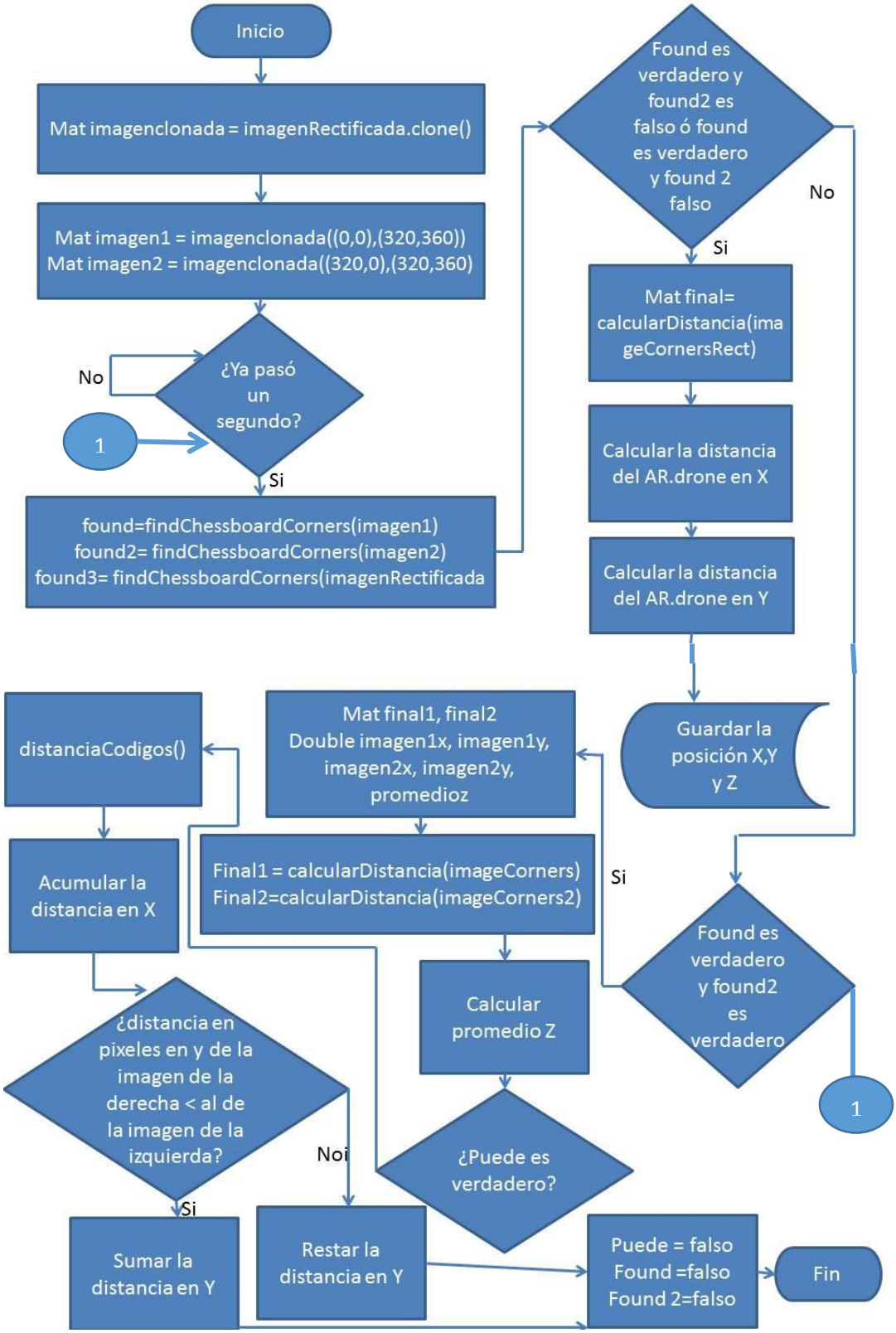


Figura 16. Diagrama de flujo función principal de slamnodistance.cpp

Luego de esto se calcula la posición real en ‘x’ y en ‘y’ del AR.drone con respecto al sistema de referencias global, esto se hace sumando la distancia que se tenga acumulada en ‘x’ y en ‘y’ entre los códigos que se han ido observando en la trayectoria, es decir, realizar el desplazamiento de los ejes de coordenadas entre los códigos que se han observado, después de este proceso y de tener la posición del AR.drone con respecto al mundo en ‘x’, ‘y’ y ‘z’ se almacena en un archivo, donde se guarda las posiciones a lo largo del todo el desplazamiento.

El siguiente *if()* que se encuentra es donde se pregunta si se encontró esquinas solamente en la mitad de la imagen derecha, es decir que found esté en falso y found1 en verdadero y se realiza el mismo proceso explicado con el anterior *if()* ya que se tiene el mismo comportamiento donde se puede esperar que en la imagen completa solo se tenga un código.

En el *if()* donde se pregunta si los found correspondientes a la parte izquierda y derecha son verdaderos es el caso donde en la imagen se pueden observar dos códigos y es el momento donde se hace el cálculo de la distancia entre estos pero no se calcula la ubicación del AR.drone, ya que en la imagen completa solo encontraría las esquinas de uno de los códigos pero no se tendría certeza sobre cuál de los dos códigos haciendo una contradicción sobre cuál de los dos debería referenciarse.

En caso de tener los dos códigos en la imagen lo que se hace es declarar dos variables de tipo MAT llamadas final1 y final2 donde se almacenará la distancia en ‘z’ con respecto al código que se tiene en la mitad izquierda y la mitad derecha de la imagen respectivamente, para esto se usa la función *calcularDistancia()*, pasándole como parámetros las esquinas de la parte izquierda y derecha respectivamente, como se explicó antes los valores que devuelve esta función al usar *solvePnP()* no serán los reales al no pasarle la imagen completa pero en cuanto al valor que se devuelve de la profundidad si es muy cercano por lo que se usa solo este parámetro de lo que devuelve dicha función, como el valor que devuelve esta función corresponde a cada una de las partes de la imagen se saca un promedio de la distancia y es almacenado en la variable promedioz que es de tipo double.

Para calcular la distancia entre los códigos se debe tener la ubicación en píxeles del origen de los dos códigos que se están observando es para esto que se declaran cuatro variables de tipo double llamadas imagen1x, imagen1y, imagen2x, imagen2y en las cuales se almacenará su posición en ‘x’ y ‘y’ del punto 1 (ver imagen 9) de la parte izquierda y derecha de la imagen respectivamente, estos valores se obtienen del vector que es llenado por la función *findChessboardCorners()* para cada una de las partes.

Luego se pregunta si ‘puede’ se encuentra en verdadero, esta variable es una variable que se usa para saber si esta misma imagen se ha analizado en el segundo anterior y se usa para asegurarnos que la distancia entre dos códigos solo sea calculada una vez porque de otra manera se almacenaría la distancia de estos códigos como una distancia nueva.

Si esta variable se encuentra en verdadero se calcula la distancia entre los códigos haciendo el llamado a la función *distanciaCodigos()* y la distancia en ‘x’ es almacena haciendo una resta a lo que ya se tenía almacenado ya que se supone que los códigos se van a ir encontrando a la izquierda del anterior que de acuerdo al sistema de coordenadas es un desplazamiento en el sentido negativo del eje ‘x’.

Para almacenar la distancia en ‘y’ si se debe evaluar si al código nuevo en aparecer se encuentra arriba o abajo del código anterior ya que esto determinará si el desplazamiento se hizo en el sentido negativo del eje o en el positivo y es por esto que se implementan los siguientes dos *if()* donde se pregunta si la posición en píxeles de la parte izquierda de la imagen, que es donde se supone que apareció el código nuevo, se encuentra en un valor mayor o menor al del código de la derecha que en caso de que sea menor corresponde a un desplazamiento en el sentido negativo y si es un valor mayor será un desplazamiento en el sentido positivo.

Finalmente se vuelven a poner las variables found y found1 en falso para que en la siguiente imagen estos puedan tener un valor real de lo que devuelve *findChessboradCorners()*.

En la figura 17 se muestra el diagrama de flujo que resume el proceso que ejecuta el algoritmo.

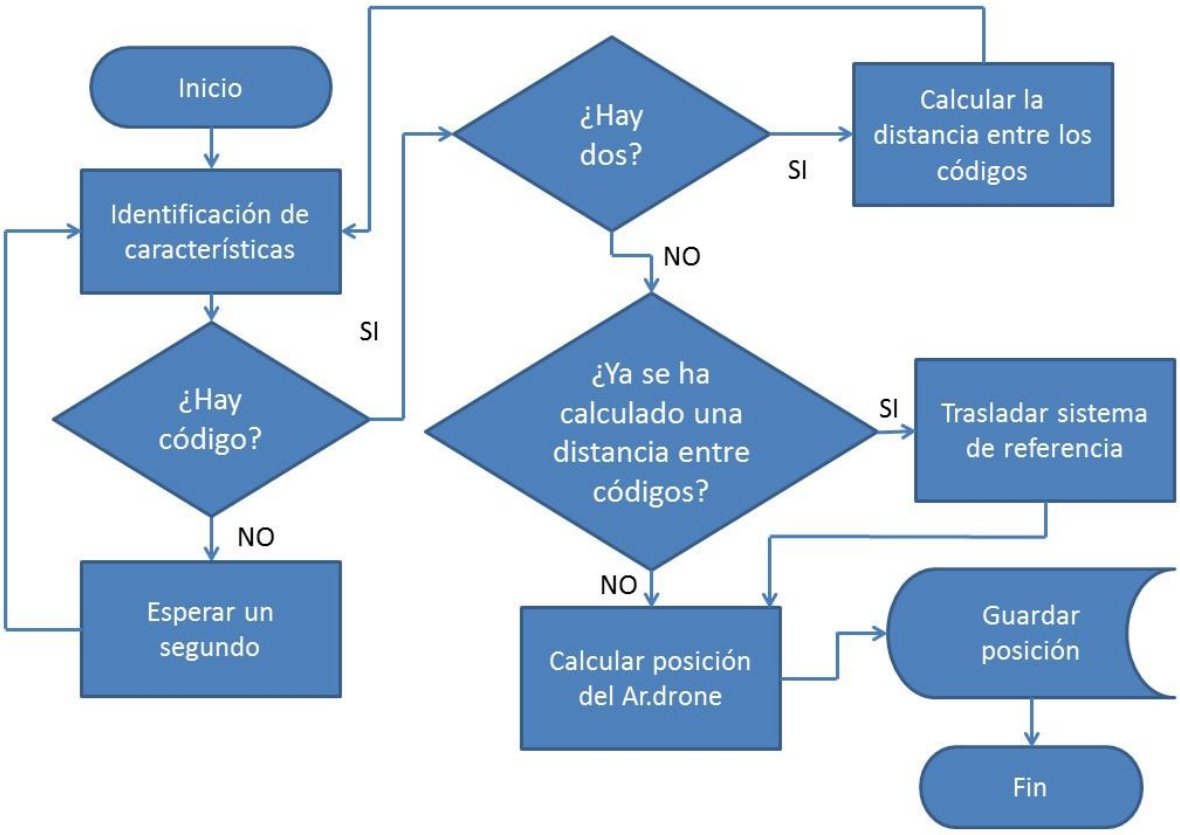


Figura 17. Diagrama de flujo algoritmo slamnodistance.

4.4 CREACIÓN DEL ENTORNO DE SIMULACIÓN

Para la creación del entorno de simulación, primero se debe crear el entorno en algún programa que permita exportar dicho entorno a un archivo con extensión ".dae", que es un formato de archivos para aplicaciones con interacción 3D.

Es por esto que se seleccionó SketchUp que es un programa de diseño gráfico y modelado en tres dimensiones basado en caras. En este programa se pueden realizar diseños complejos en 3D de forma sencilla, además de brindar una gran galería de objetos, texturas e imágenes que se pueden descargar para usar en otros diseños.

Luego de descargar este programa se comenzó con el diseño de nuestro entorno de simulación, donde la primera restricción es que debe ser un entorno interior por lo tanto contiene varias texturas y objetos que son comunes en un entorno interior.

Para tener un entorno completo y aunque no se vaya a usar en toda su extensión, para este proyecto se decidió diseñar un apartamento con todos los elementos que éste contiene como lo son cocina, sala, comedor, habitaciones, baños, etc.

Se empezó creando cada uno de los espacios que debe contener el apartamento, es decir se hizo primero la construcción de todas las paredes externas y separaciones internas de nuestro apartamento, definiendo así las áreas que tendría cada uno de los espacios.

Posteriormente a esto se comenzaron a situar los objetos característicos de cada uno de los espacios, para esto se usó la opción de SketchUp de descargar modelos ya creados como los mostrados en la figura 18, que son objetos que así tengan tamaños determinados, pueden modificarse para que se acoplen a nuestros espacios, además de esto son objetos que pueden a llegar a ser muy reales permitiendo así dar más credibilidad al entorno de simulación, que es algo que no se podría hacer si cada uno de los objetos que contiene el entorno lo hubiésemos creado nosotros.

Figura 18. Objetos SketchUp

Luego de todo este proceso, el escenario que se obtuvo en SketchUp es el mostrado en la figura 19, donde se puede observar todo el apartamento con ciertos objetos en su cocina, sala, comedor y habitaciones, además de texturas en el piso y algunas paredes que son elementos suficientes para el entorno ya que el único espacio que se usará es el comprendido entre la sala y el comedor, es decir la pared que comparten están dos áreas para ubicar los códigos.

Figura 19. Modelo del apartamento en SketchUp

Cuando ya se tiene el entorno que se desea, se debe ir a la opción que brinda SketchUp para exportar el archivo a un tipo de archivo de COLLADA (.dae) que es el archivo que se usa para ser agregado el entorno a ROS.

Al exportar el archivo se crea el archivo de COLLADA también se crea una carpeta donde están guardadas las texturas que se usan en todo en el entorno, el archivo como la carpeta hay que agregarlos a ROS para que este muestre el entorno como es porque se crea un error al no contar con las texturas ya que el archivo de COLLADA hace un llamado a cada una de estas texturas.

Ahora, en el entorno de trabajo de ROS que es el lugar donde se tienen las carpetas de los paquetes como lo son el "ardrone_autonomy", "tum_simulator" y "tum_ardrone" se debe ingresar a la carpeta correspondiente al paquete de "tum_simulator" para agregar lo que se creó con SketchUp, tanto la carpeta como el archivo de COLLADA deben agregarse en la carpeta "models" ubicada dentro de este paquete en la carpeta "cvg_sim_gazebo/Media".

También es necesario crear un archivo con extensión ".launch" pero con el mismo nombre del entorno y es un archivo de configuración escrito en XML para ejecutar varios nodos a la vez, en este archivo se deben especificar que nodos se deben ejecutar, los argumentos de entrada a cada uno de estos nodos y los parámetros necesarios para crear una aplicación completa, en nuestro caso este archivo ejecuta el nodo del simulador para nuestro escenario y el nodo correspondiente a todo el modelo de AR.drone que ya se encuentra implementado. Este archivo debe agregarse en la ruta "tum_simulator/cvg_sim_test/launch".

Otro archivo que se debe crear es un archivo con extensión ".world" correspondiente al entorno, este tipo de archivo también se escribe en XML y en el se deben especificar parámetros del motor físico ODE (como el tiempo de integración, el valor de la gravedad, entre otros), del motor gráfico OGRE (renderizado, sombras, cielo, ambiente, etc) y de los objetos que deben aparecer en el simulador automáticamente. Es importante resaltar que en este archivo también se especifica la reducción que se desea que tenga el entorno, el AR.drone y cada uno de los componentes que se tengan en el simulador se debe resaltar que las medidas de los objetos y de los componentes de la simulación deben darse en pulgadas, es por esto que en nuestro archivo "apto.world" se tuvo que realizar una reducción del entorno de metros a pulgadas. Este archivo debe agregarse a la ruta "tum_simulator/cvg_sim_gazebo".

Luego de crear cada uno de estos archivos y agregarlos a sus respectivas rutas se puede hacer el llamado al simulador desde el terminal con la siguiente línea de comando:

roslaunch cvg_sim_test apto.launch

Obteniendo como resultado la figura 20, donde se muestra el entorno de simulación creado con el AR.drone dentro de este y ejecutado desde ROS.

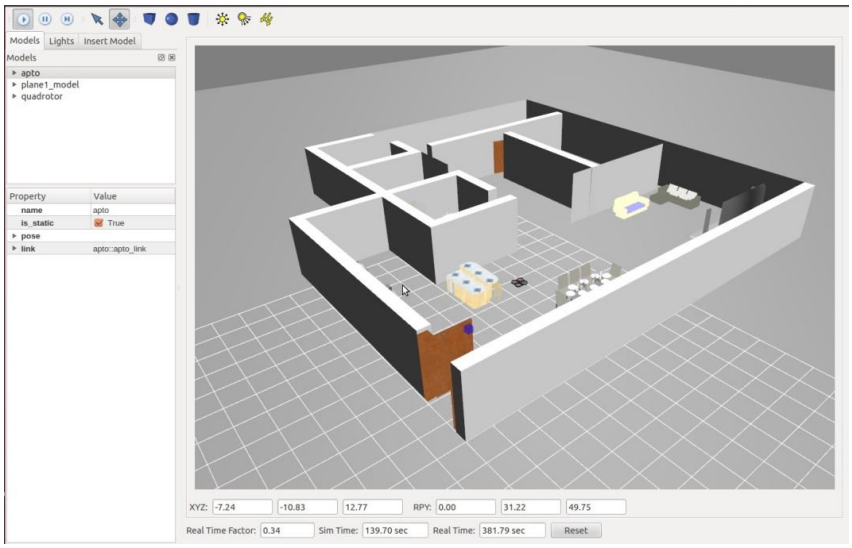


Figura 20. Entorno de simulación corriendo desde ROS

5. ANÁLISIS DE RESULTADOS

En esta sección se explicarán las pruebas realizadas para medir la precisión de cada uno de los dos algoritmos desarrollados. Para evaluar el primer algoritmo se realizaron cinco pruebas, y para la evaluación del segundo se realizaron cuatro.

Cada una de las pruebas presentadas a continuación tiene como objetivo comparar las coordenadas 'x', 'y' y 'z' obtenidas mediante la ejecución de los algoritmos con la ubicación real (ground truth) referida al mismo sistema de coordenadas midiéndolas con un flexómetro.

5.1 PRUEBAS ALGORITMO SLAMMONOCULAR.CPP

Las primeras pruebas realizadas se hicieron para medir la efectividad del algoritmo slammonocular.cpp, para esto se realizaron pruebas bajo diferentes condiciones de visibilidad de los códigos con el fin de determinar el error dependiendo de la posición desde la cual es visto cada uno de los tableros de ajedrez.

En cada una de las figuras que muestran el comportamiento de las muestras tomadas se grafica bajo el nombre de 'lineal' la tendencia lineal que toma dicho comportamiento.

5.1.1 PRIMERA PRUEBA

Esta primera prueba consistió en ubicar el AR.drone de tal forma que el código visto quedara centrado en la imagen captada por su cámara frontal, se dejó por un periodo de tiempo en el cual se guardaron los datos de posición registrados por el algoritmo.

Los resultados se muestran a continuación en la tabla 4.

PRIMERA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X (m)	0,105	X (m)	0,0646	0,0404	38,5%
Y (m)	0,17	Y (m)	0,1704	0,0004	0,2%
Z (m)	-1,02	Z (m)	-1,0378	0,0178	1,7%
Total Muestras	45			*Absoluto(m)*	*Relativo*

Tabla 4. Resultados prueba 1 slammonocular.cpp

A continuación se muestra la forma en la cual están distribuidos los datos obtenidos para cada una de las coordenadas.

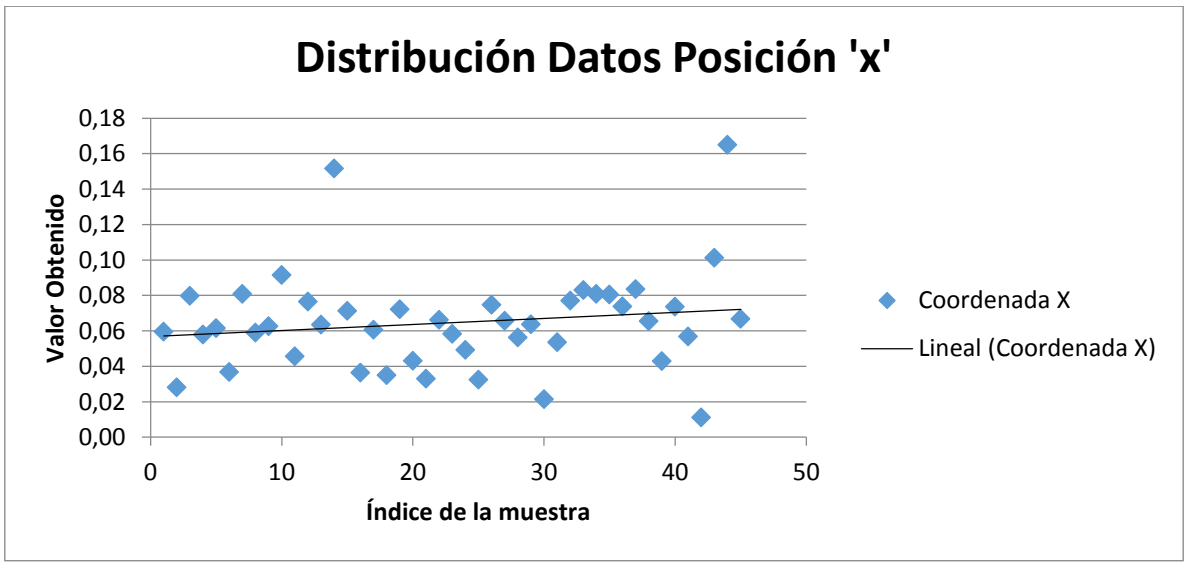


Figura 21. Distribución de datos posición 'x' para la primera prueba. Desviación estándar 0.028.

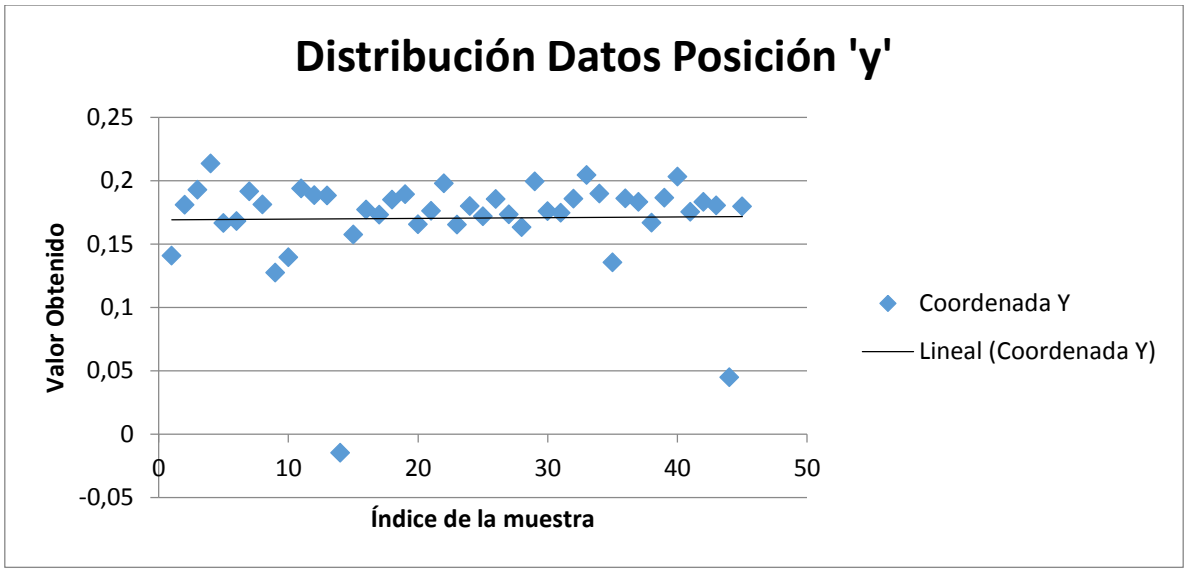


Figura 22. Distribución de datos posición 'y' para la primera prueba. Desviación estándar 0.038.

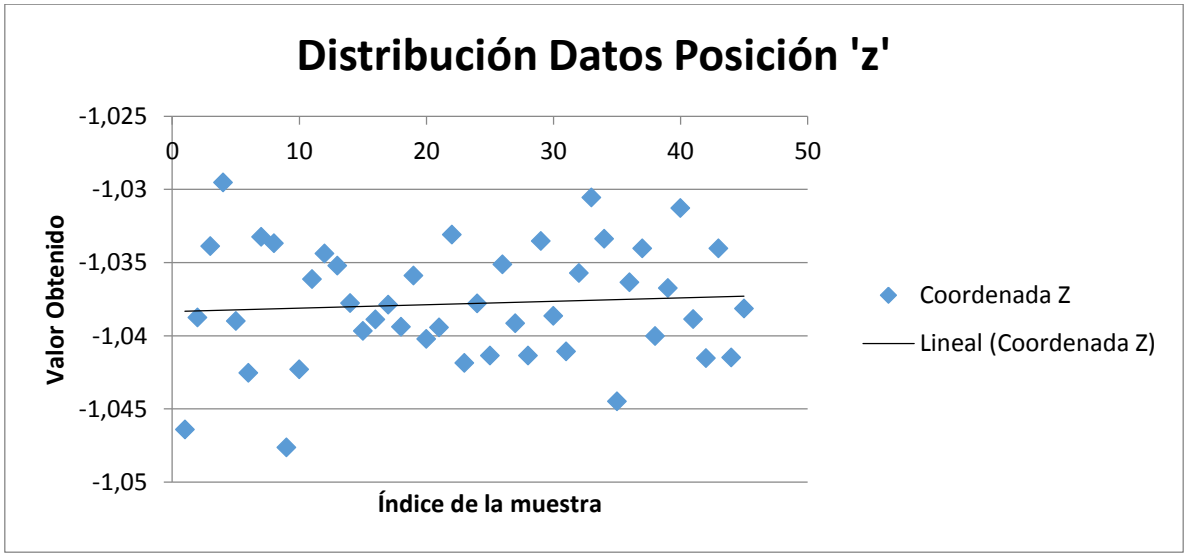


Figura 23. Distribución de datos posición 'z' para la primera prueba. Desviación estándar 0.004.

Como se muestra en la tabla y diferentes figuras que integran esta sección se observa de manera clara que en términos generales el error que presenta el algoritmo para esta vista del código no es muy bajo en promedio, sin embargo los datos obtenidos para la posición 'y' y 'z' son bastante cercanos al valor real. Cabe destacar adicionalmente que hay casos para las coordenadas 'x' y 'y' en las que los datos obtenidos difieren completamente de los datos reales, sin embargo estos datos representan entre el 6% y 7% de la cantidad total de datos. Este fenómeno se presenta debido a la inestabilidad de la conexión WiFi entre la estación base y el AR.drone.

5.1.2 SEGUNDA PRUEBA

La segunda prueba realizada consistió en ubicar el AR.drone de tal manera que el código observado quedara en la parte derecha de la imagen capturada, es decir, el AR.drone está viendo el código desde el costado izquierdo. Se dejó el AR.drone en esta misma posición por un tiempo determinado y se registraron todos los datos obtenidos, arrojando los resultados que se muestran a continuación.

SEGUNDA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X (m)	-0,33	X (m)	-0,3999	0,0699	21,2%
Y (m)	0,17	Y (m)	0,1878	0,0178	10,5%
Z (m)	-1	Z (m)	-0,991	0,009	0,9%
Total Muestras	62			*Absoluto(m)*	*Relativo*

Tabla 5. Resultados prueba 2 slammonocular.cpp

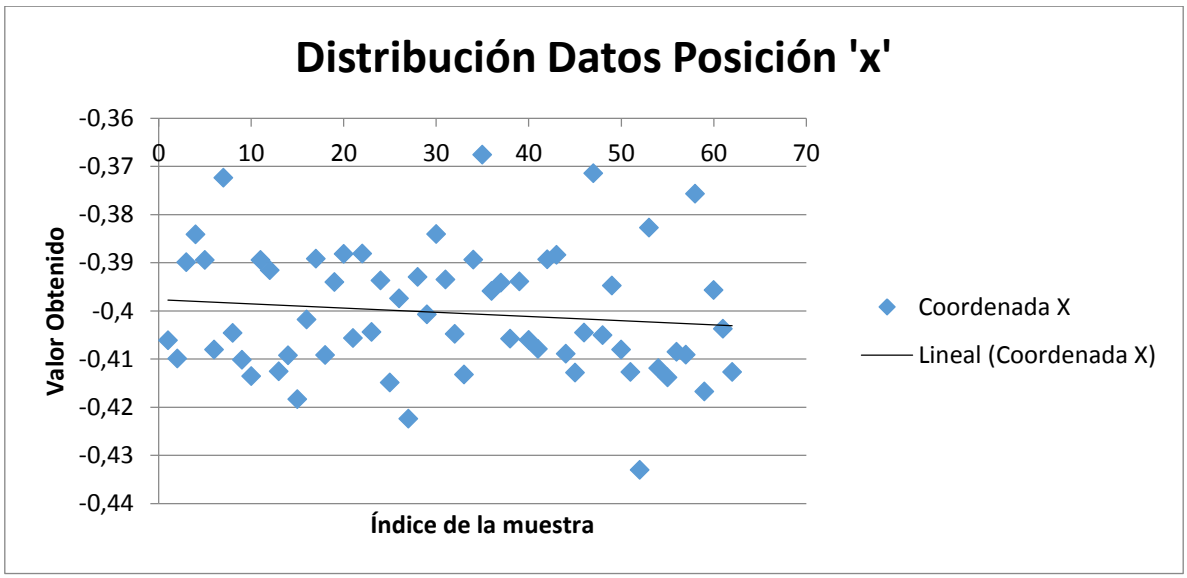


Figura 24. Distribución de datos posición 'x' para la segunda prueba. Desviación estándar 0.013.

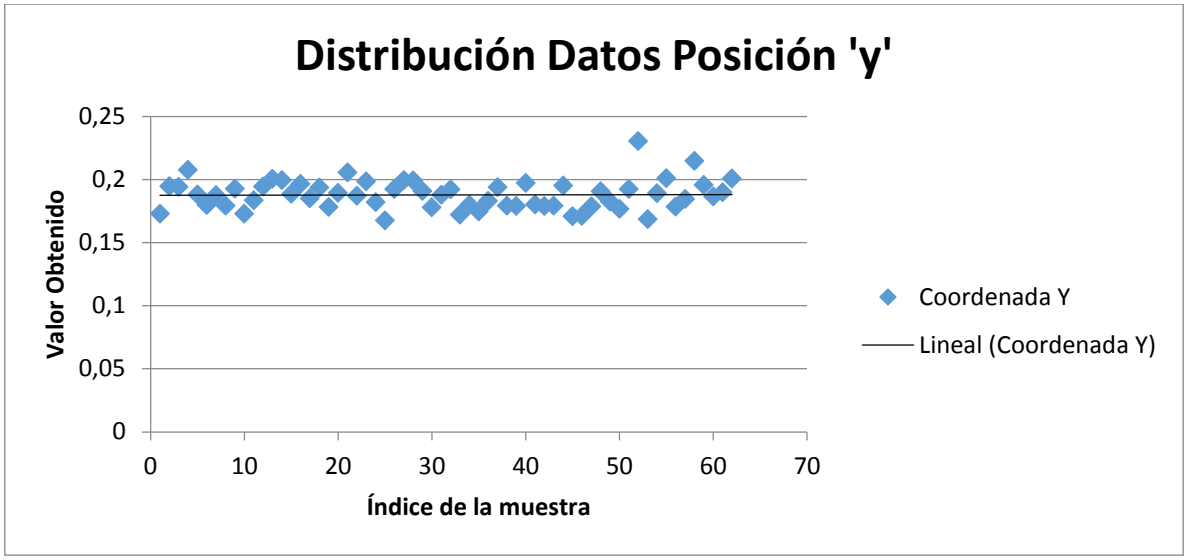


Figura 25. Distribución de datos posición 'y' para la segunda prueba. Desviación estándar 0.011.

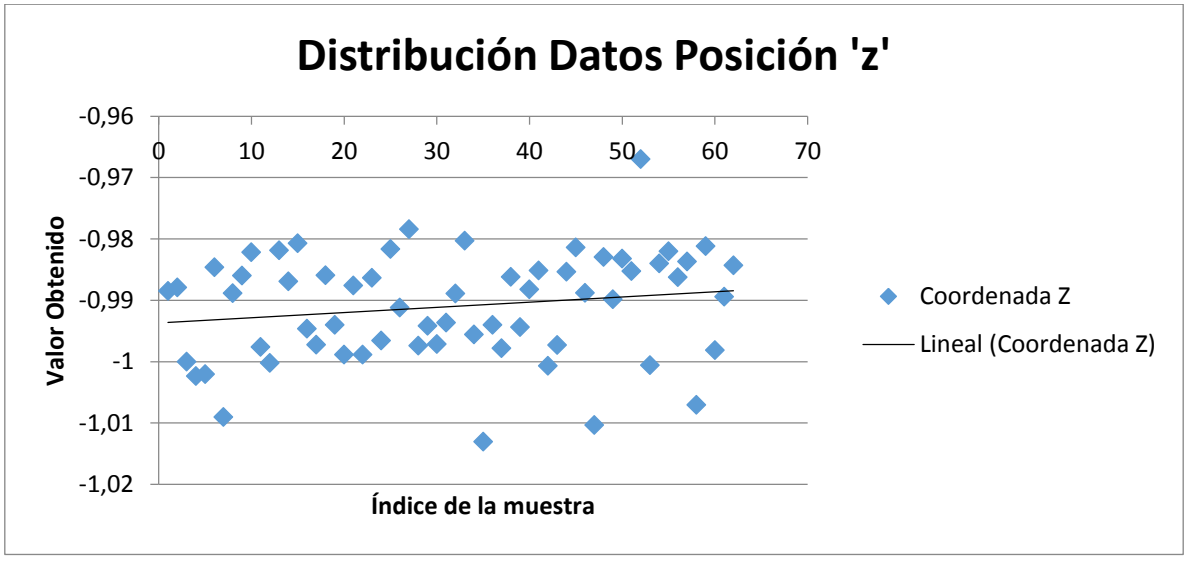


Figura 26. Distribución de datos posición 'z' para la segunda prueba. Desviación estándar 0.008.

Comparando los resultados de esta prueba con los de la anterior se mantiene el porcentaje de error más alto sobre la coordenada 'x'. En general la distribución de los datos para las tres coordenadas es uniforme y su media tiende a permanecer constante a lo largo del tiempo.

5.1.3 TERCERA PRUEBA

La tercera prueba consistió en ubicar el AR.drone de tal manera que en la imagen capturada por su cámara frontal el código quedara ubicado en el lado izquierdo de la misma, es decir el AR.drone está observando el código desde el costado derecho. Se guardaron los datos registrados por el algoritmo para su análisis.

TERCERA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	0,51	X	0,5058	0,0042	0,8%
Y	0,17	Y	0,1899	0,0199	11,7%
Z	-0,92	Z	-0,9021	0,0179	1,9%
Total Muestras	56			*Absoluto(m)*	*Relativo*

Tabla 6. Resultados prueba 3 slammonocular.cpp

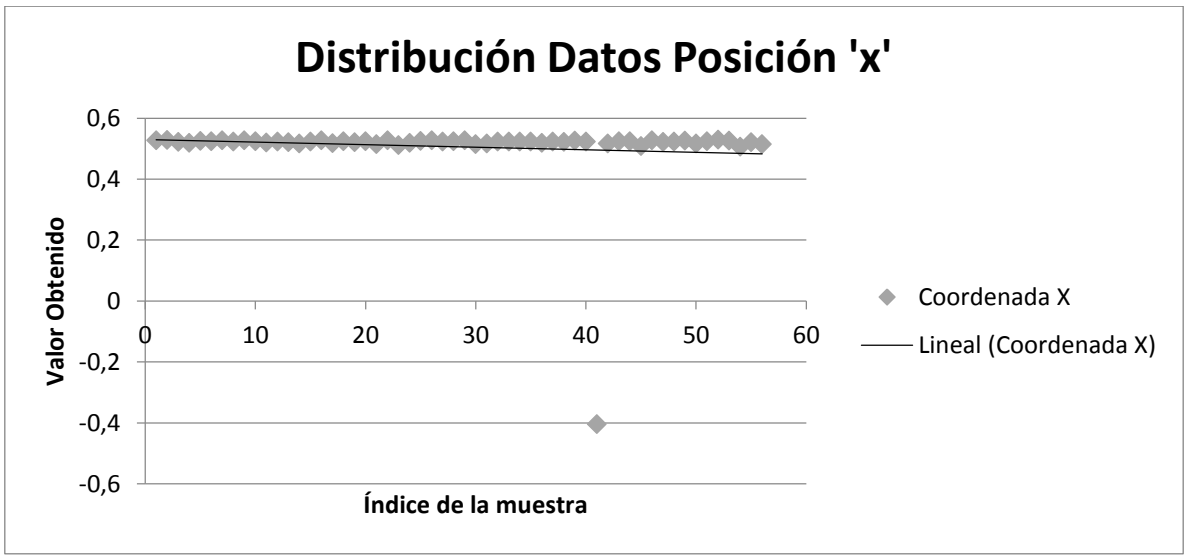


Figura 27. Distribución de datos posición 'x' para la tercera prueba. Desviación estándar 0.123.

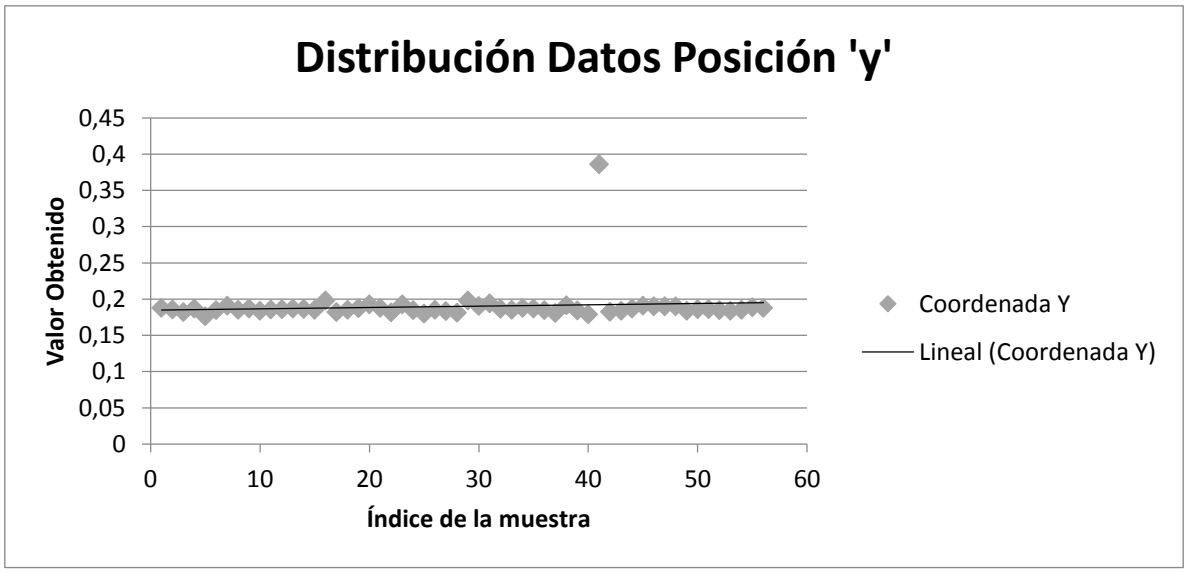


Figura 28. Distribución de datos posición 'y' para la tercera prueba. Desviación estándar 0.027.

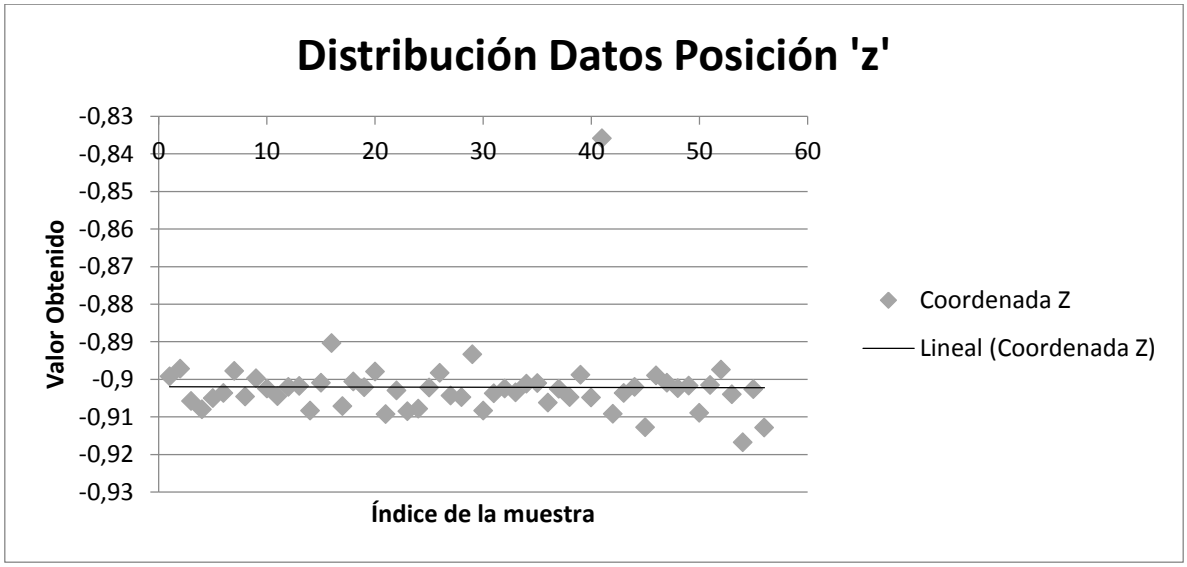


Figura 29. Distribución de datos posición 'z' para la tercera prueba. Desviación estándar 0.01.

En este caso, contrario a los resultados obtenidos en las anteriores dos pruebas, el porcentaje de error más alto se presenta en la coordenada 'y'. En promedio el error es bajo y se puede afirmar que los datos son los más homogéneos de los obtenidos hasta el momento. Existe un dato sobre la muestra número 40 que difiere del comportamiento normal de la tendencia, mostrando el efecto de la inestabilidad de la conexión WiFi.

5.1.4 CUARTA PRUEBA

La cuarta prueba para la evaluación del primer algoritmo consistió en cambiar la verticalidad de ver el código incluyendo un ángulo de vista de 22° con respecto al eje 'z'. Es decir el AR.drone está ubicado en el costado izquierdo con respecto a un código con un ángulo de inclinación.

CUARTA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	-0,34	X	-0,3769	0,0369	10,9%
Y	0,17	Y	0,1925	0,0225	13,2%
Z	-0,84	Z	-0,8278	0,0122	1,5%
Total Muestras	52			*Absoluto(m)*	*Relativo*

Tabla 7. Resultados prueba 4 slammonocular.cpp

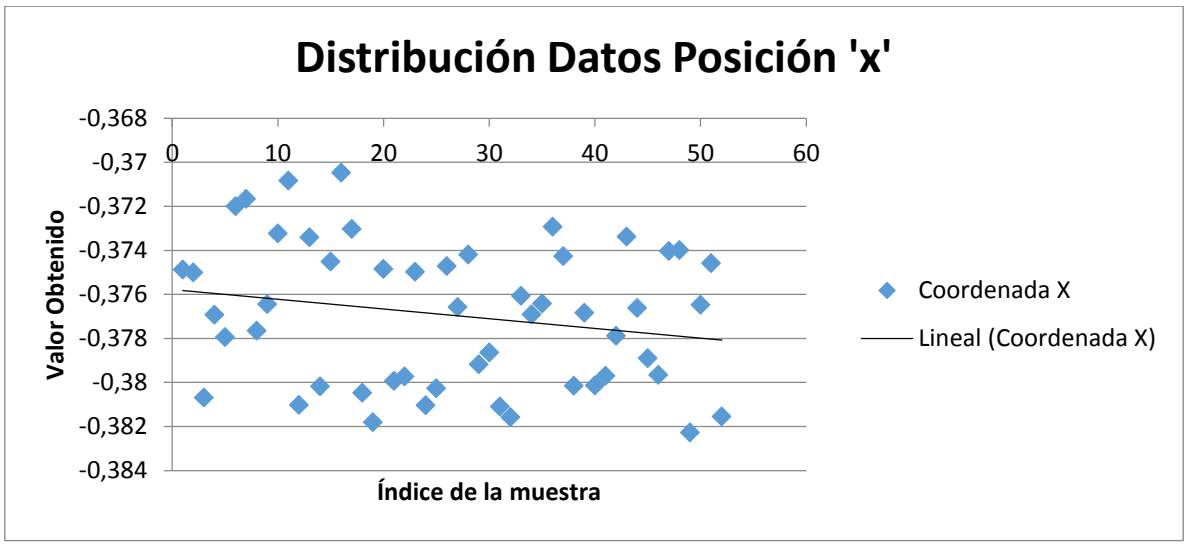


Figura 30. Distribución de datos posición 'x' para la cuarta prueba. Desviación estándar 0.003.

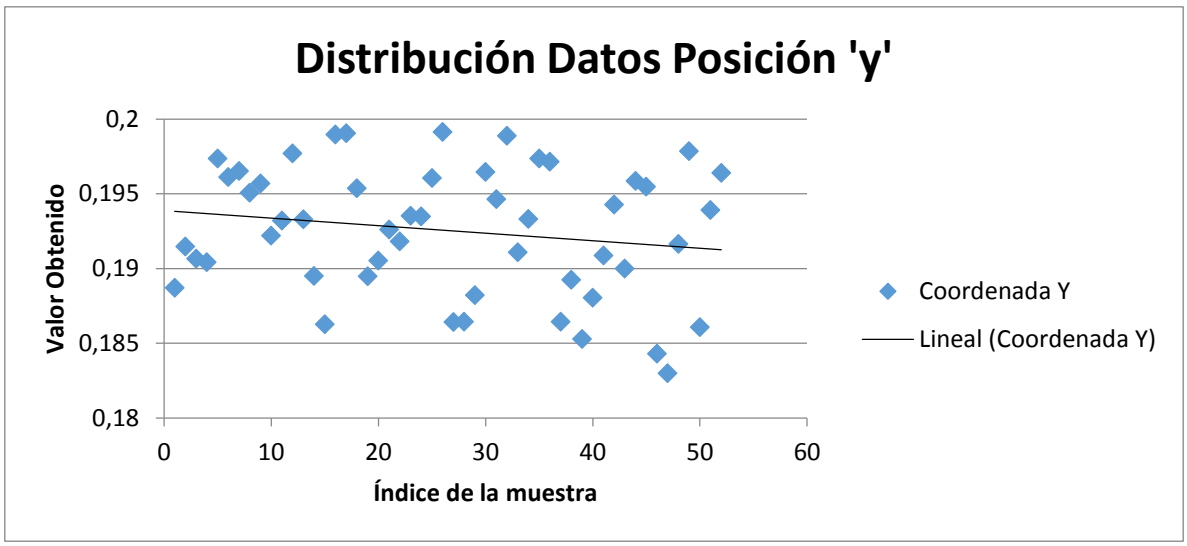


Figura 31. Distribución de datos posición 'y' para la cuarta prueba. Desviación estándar 0.004.

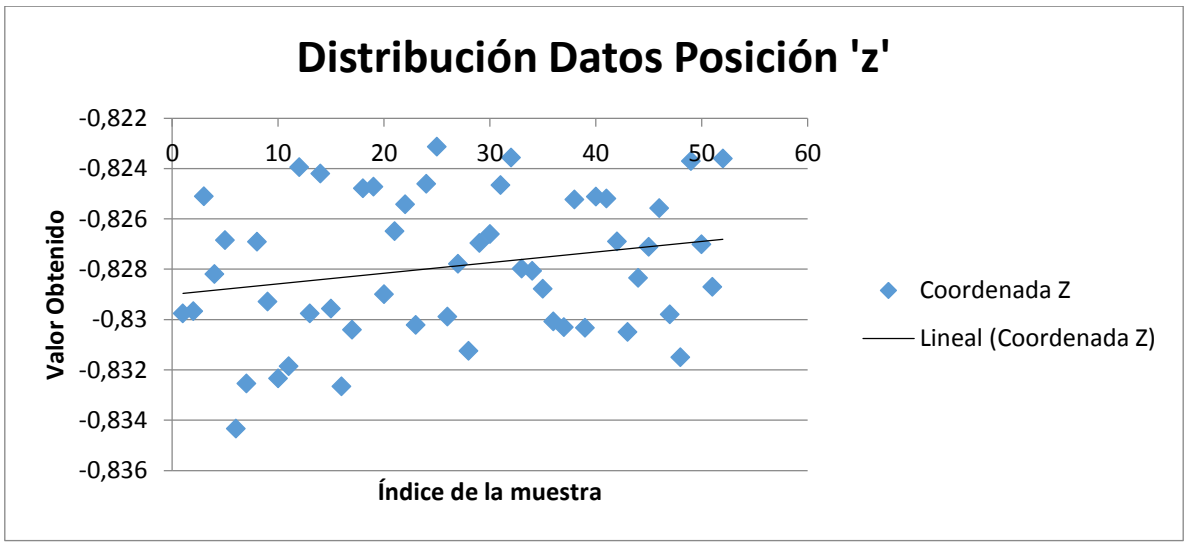


Figura 32. Distribución de datos posición 'z' para la cuarta prueba. Desviación estándar 0.002.

Como se puede observar en la tabla 7, el error promedio aumenta de nuevo comparado con la tercera prueba, esto indica que el algoritmo en general es mucho más preciso cuando el AR.drone está observando el código desde el lado derecho del mismo con una vista completamente frontal. En términos de la distribución de datos se puede decir que se presentan de una manera homogénea, soportado en la baja desviación estándar, lo que quiere decir que el algoritmo no es muy bueno bajo estas condiciones.

5.1.5 QUINTA PRUEBA

La quinta y última prueba realizada al primer algoritmo consistió en ubicar el AR.drone de manera paralela al código y no perpendicular como se había venido probando anteriormente, los resultados son mostrados y resumidos en la tabla 8.

QUINTA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	-1,15	X	-1,1412	0,0088	0,8%
Y	0,17	Y	0,1722	0,0022	1,3%
Z	-0,43	Z	-0,4125	0,0175	4,1%
Total Muestras	48			*Absoluto (m)*	*Relativo*

Tabla 8. Resultados prueba 5 slammonocular.cpp

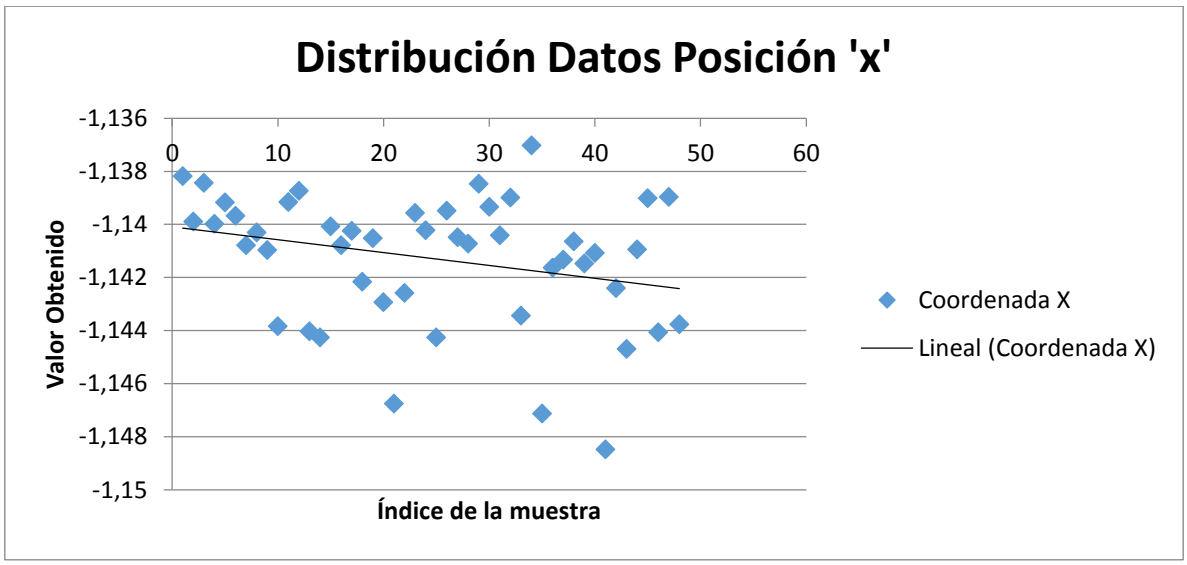


Figura 33. Distribución de datos posición 'x' para la quinta prueba. Desviación estándar 0.002.

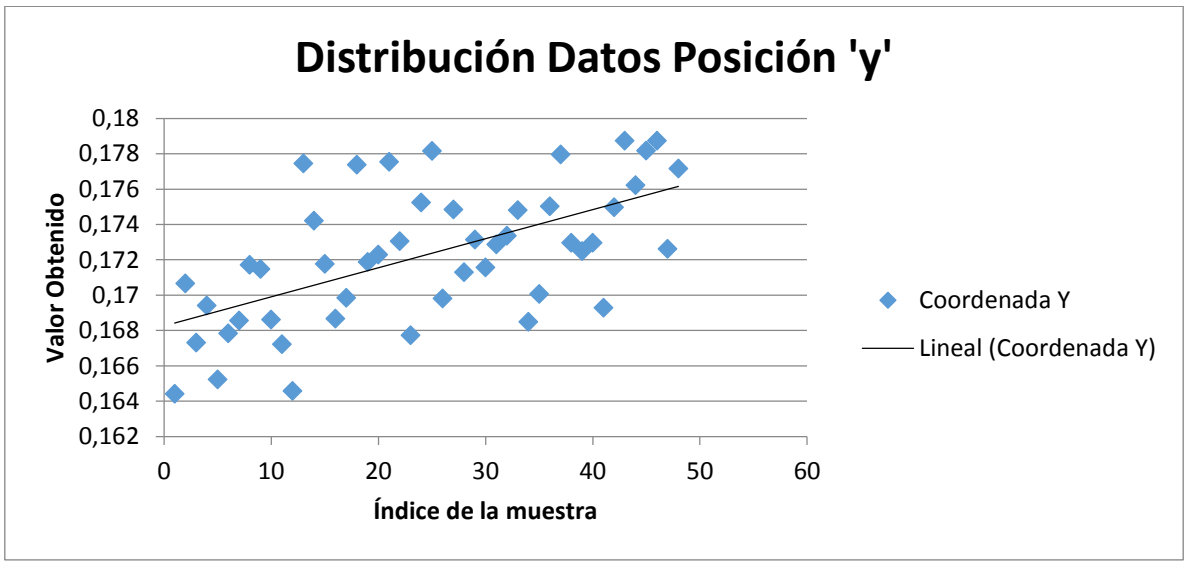


Figura 34. Distribución de datos posición 'y' para la quinta prueba. Desviación estándar 0.003.

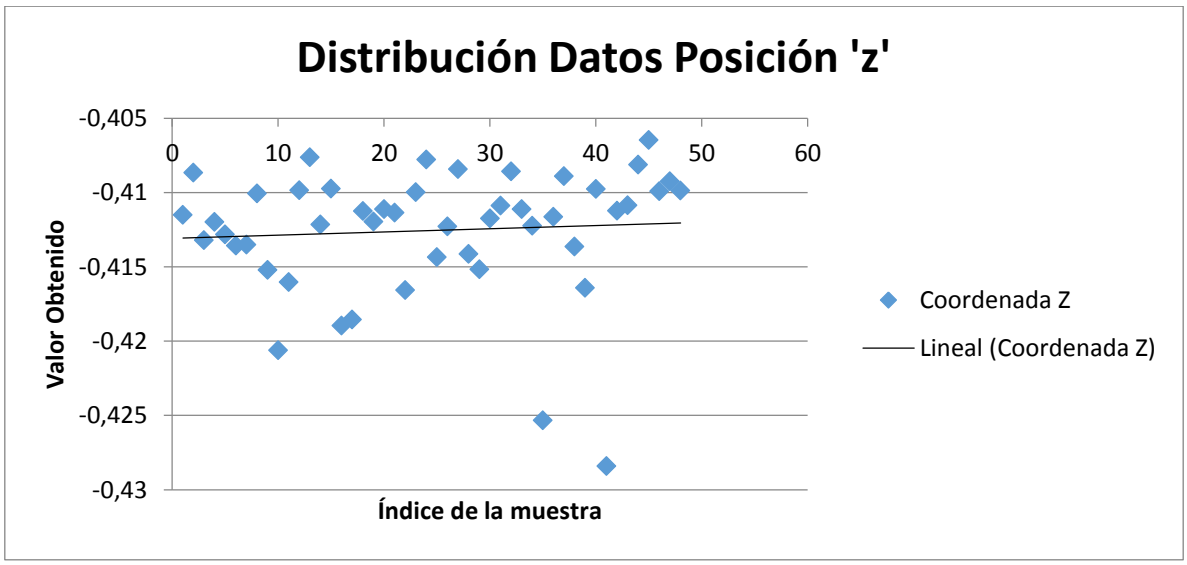


Figura 35. Distribución de datos posición 'z' para la quinta prueba. Desviación estándar 0.004.

De las pruebas realizadas hasta el momento, esta es la que presenta un porcentaje promedio de error más bajo, las muestras presentan una desviación estándar baja, sin embargo como en casos anteriores se presentan casos no frecuentes en los que la muestra se aleja de manera considerable de la tendencia.

Figura 36. Secuencia de códigos identificados al ejecutar slammonocular.cpp

En la figura 36 se observa la secuencia de códigos capturados por la cámara frontal del AR.drone en una ejecución del algoritmo slammonocular.cpp. El orden de secuencia es de izquierda a derecha y de arriba hacia abajo, es decir esta es la forma en la cual deben ordenarse los códigos para un correcto funcionamiento del algoritmo.

5.2 PRUEBAS ALGORITMO SLAMNODISTANCE.CPP

Para probar el algoritmo slamnodistance.cpp se realizaron un total de cuatro pruebas en las que se busca evaluar el desempeño del mismo bajo escenarios diferentes de lo que sería un vuelo normal.

5.2.1 PRIMERA PRUEBA

En la primera prueba se ubicó el AR.drone de tal manera que se observara el primer código de lo que sería un vuelo normal y se registraron los datos arrojados por el algoritmo correspondientes a este primer código.

PRIMERA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	-0,27	X	-0,3344	0,0644	23,9%
Y	0,15	Y	0,1499	1E-04	0,1%
Z	-1,13	Z	-1,1024	0,0276	2,4%
Total Muestras	63			*Absoluto(m)*	*Relativo*

Tabla 9. Resultados prueba 1 slamnodistance.cpp

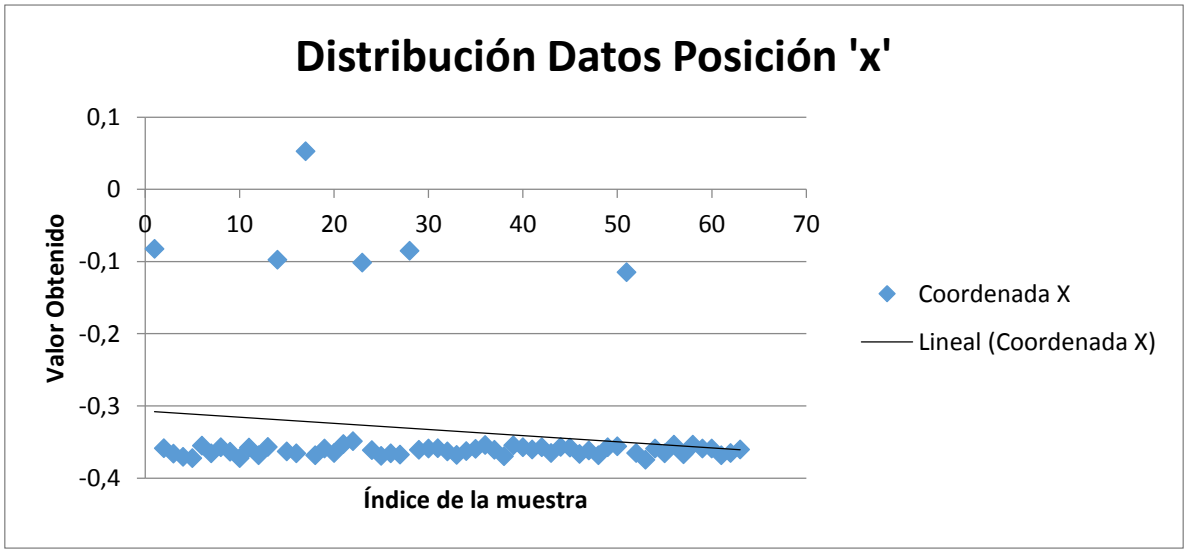


Figura 37. Distribución de datos posición 'x' para la primera prueba. Desviación estándar 0.08.

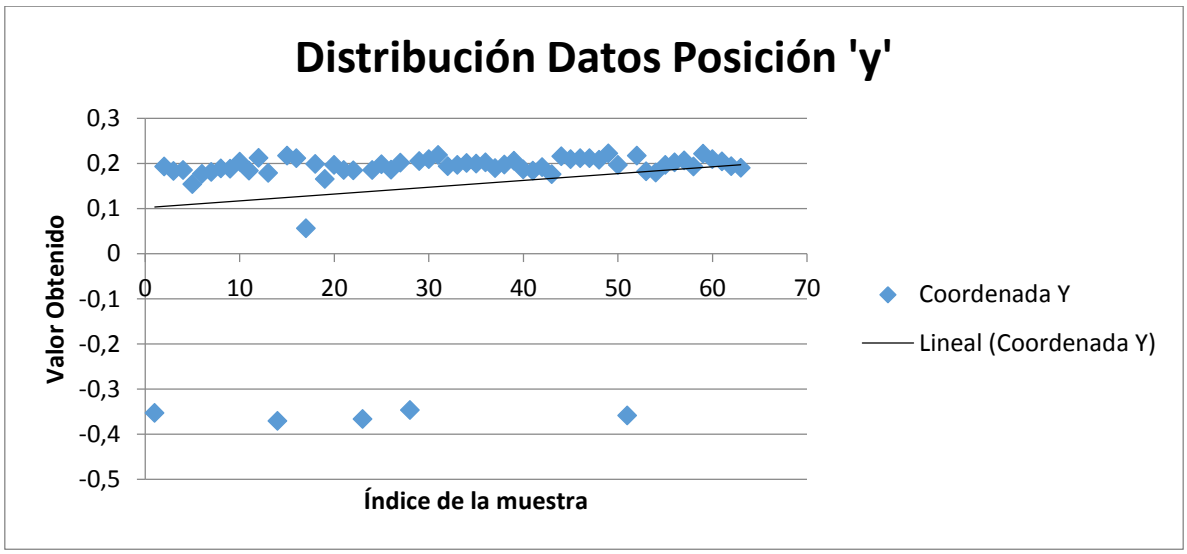


Figura 38. Distribución de datos posición 'y' para la primera prueba. Desviación estándar 0.15.

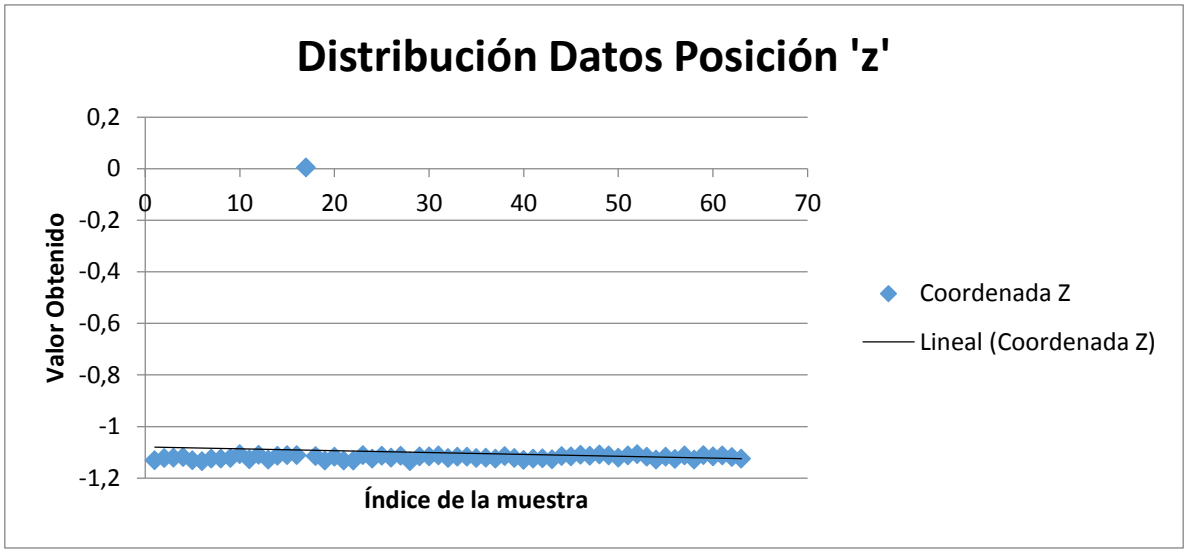


Figura 39. Distribución de datos posición 'z' para la primera prueba. Desviación estándar 0.14.

Esta prueba es exactamente la misma primera prueba del primer algoritmo. Como podemos observar el comportamiento del segundo algoritmo es muy similar: se presenta un error considerable en el eje 'x'. Adicionalmente la principal diferencia es que la homogeneidad de los datos disminuye, ya que se presentan una mayor cantidad de muestras que se salen de la tendencia normal del algoritmo, por esto su desviación estándar aumenta de manera considerable.

5.2.2 SEGUNDA PRUEBA

La segunda prueba busca evaluar una función en particular del algoritmo slamnodistance.cpp: la medición de la distancia entre códigos, para ello se ubicó el AR.drone de tal forma que en la imagen capturada se

vieran todo el tiempo dos códigos y se calculara su distancia de manera permanente. Los resultados se muestran a continuación.

SEGUNDA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	0,93	X	0,9404	0,0104	1,12%
Y	0,44	Y	0,466	0,026	5,91%
Total Muestras	66			*Absoluto(m)*	*Relativo*

Tabla 10. Resultados prueba 2 slamnodistance.cpp

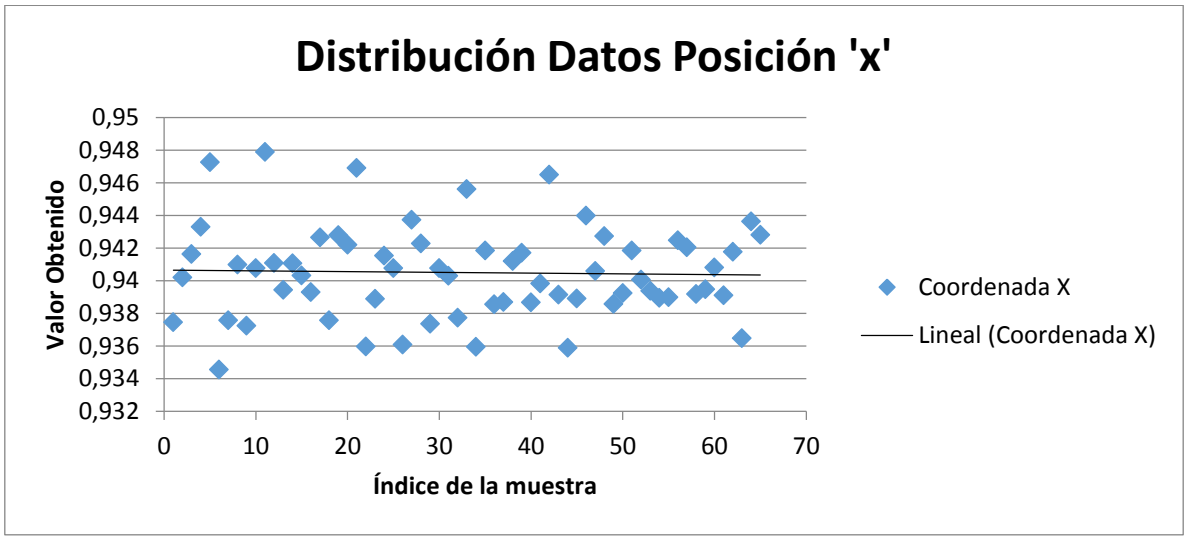


Figura 40. Distribución de datos posición 'x' para la segunda prueba. Desviación estándar 0.002.

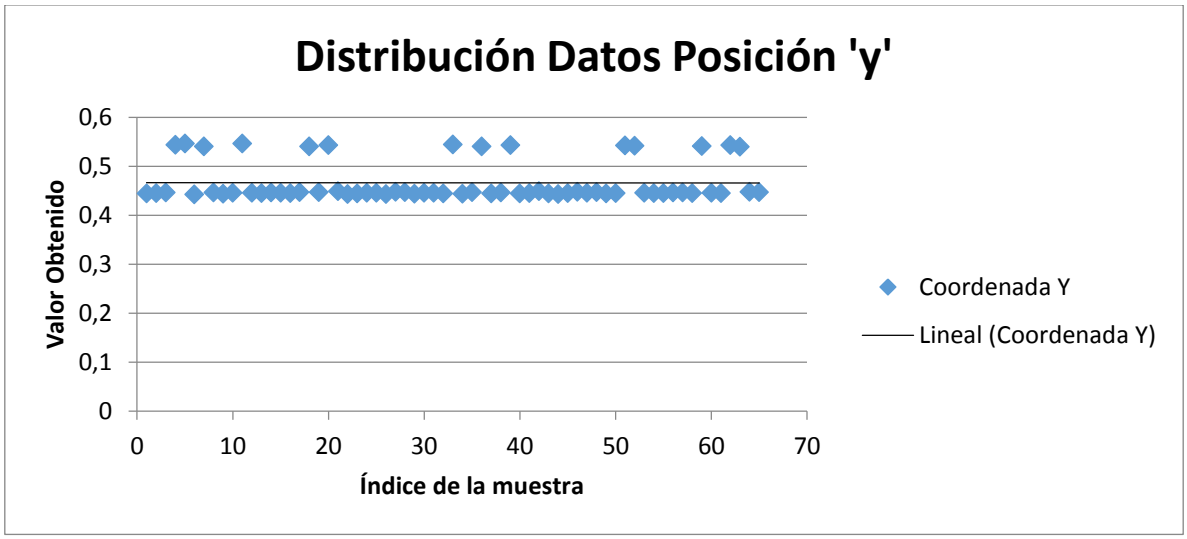


Figura 41. Distribución de datos posición 'y' para la segunda prueba. Desviación estándar 0.04.

Como se puede observar en la tabla 10 y figuras 40 y 41, el porcentaje de error presentado es realmente bajo y adicionalmente presenta una mayor homogeneidad de las muestras comparada con la prueba

anterior. Con base en esto podemos afirmar que la función mediante la cual se calcula la distancia entre dos códigos tiene un grado alto de confiabilidad.

5.2.3 TERCERA PRUEBA

La tercera prueba tiene como objetivo medir la precisión con la cual se obtienen las coordenadas del AR.drone después de haber visto 2 códigos, es decir se inició viendo un primer código, posteriormente vió dos códigos (uno encima del otro) con el fin de calcular su distancia y finalmente se registraron los datos calculados por el algoritmo para el segundo código observado.

TERCERA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	-1,21	X	-1,2919	0,0819	6,77%
Y	-0,08	Y	-0,0795	0,0005	0,63%
Z	-1,28	Z	-1,2881	0,0081	0,63%
Total Muestras	67			*Absoluto(m)*	*Relativo*

Tabla 11. Resultados prueba 3 slamnodistance.cpp

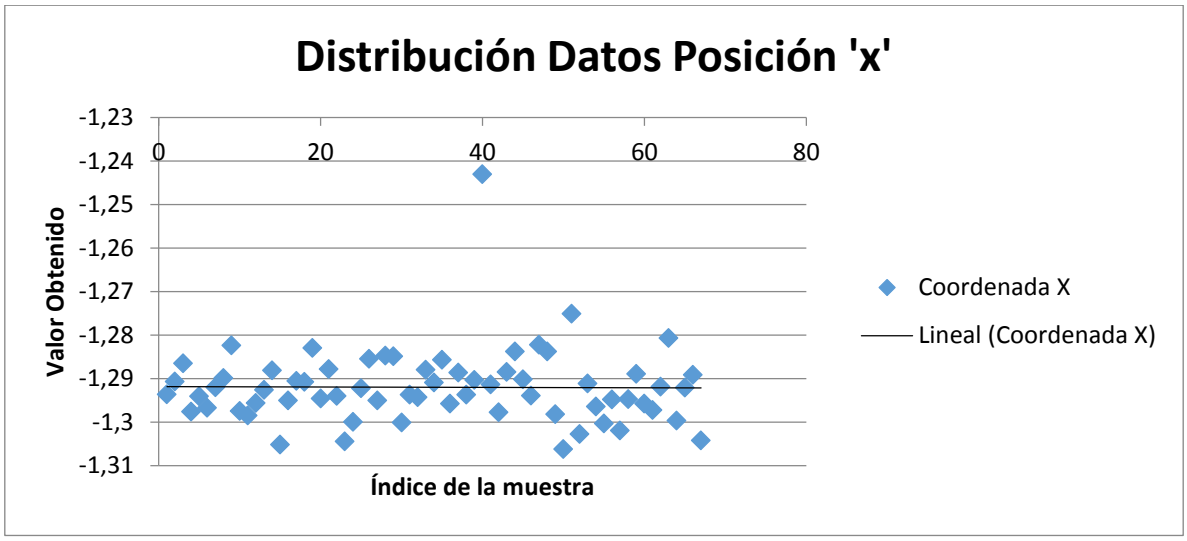


Figura 42. Distribución de datos posición 'x' para la tercera prueba. Desviación estándar 0.008.

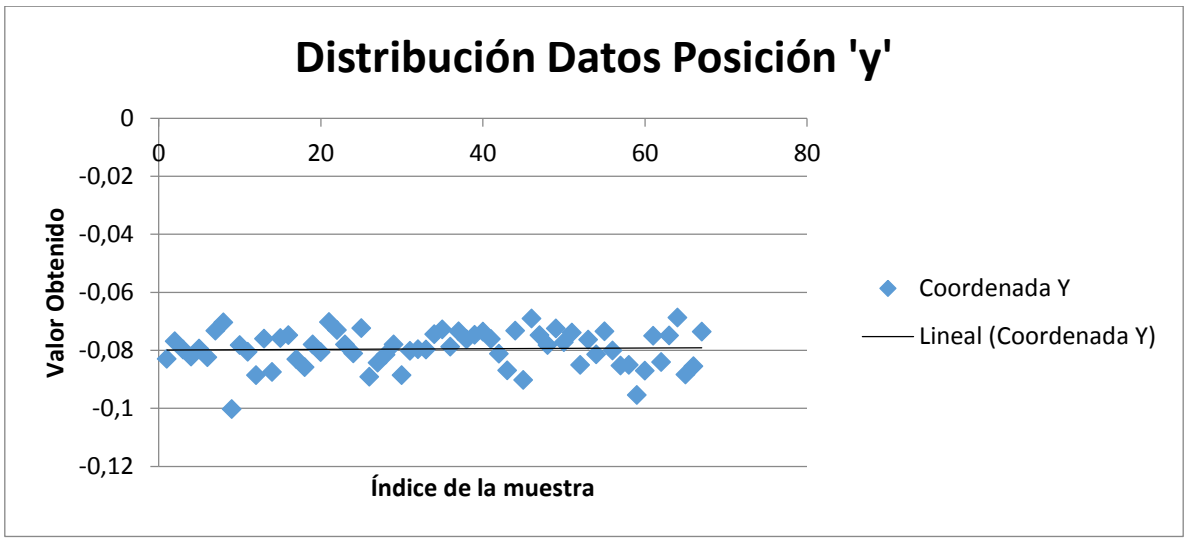


Figura 43. Distribución de datos posición 'y' para la tercera prueba. Desviación estándar 0.006.

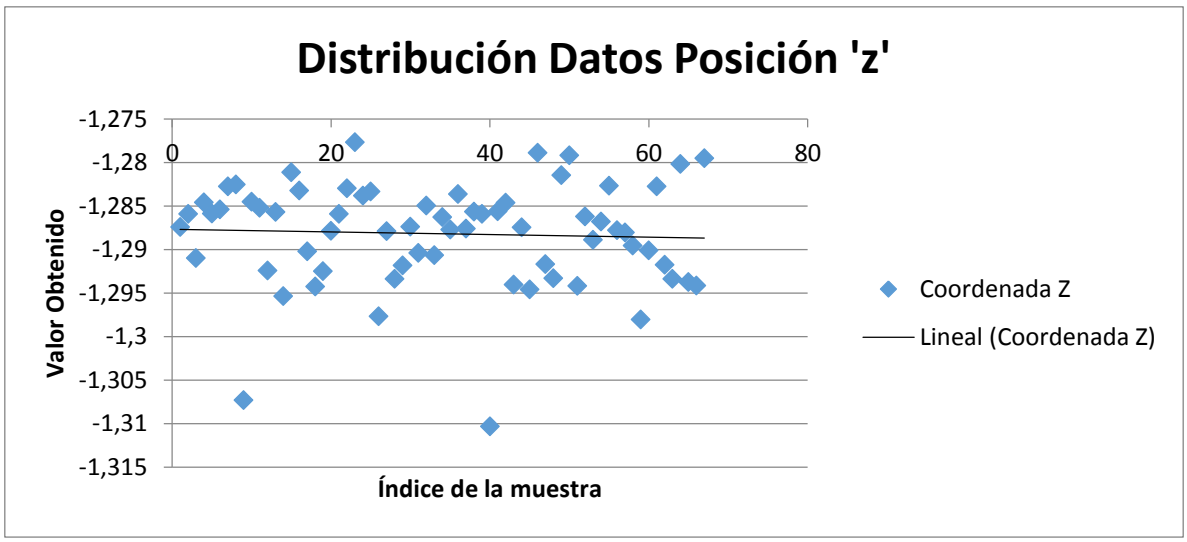


Figura 44. Distribución de datos posición 'z' para la tercera prueba. Desviación estándar 0.005.

Los resultados para esta prueba arrojan un porcentaje bajo de error, con una desviación estándar promedio muy baja, lo que muestra estabilidad en ese momento de la conexión WiFi y un cálculo bastante acertado de la posición del AR.drone.

5.2.4 CUARTA PRUEBA

La última prueba de esta serie que tienen como objetivo evaluar el desempeño del algoritmo slamnodistance.cpp tiene como objetivo ver cómo se comporta el algoritmo cuando la distancia en 'y' entre dos códigos es bastante pequeña y están ubicados de manera horizontal, es decir uno al lado del otro. En esta prueba se inició el algoritmo mientras el AR.drone observaba un primer código, luego se fue moviendo de manera horizontal hacia la izquierda permitiendo que calculara la distancia entre los códigos

y finalmente registrando los datos de posición obtenidos mientras se observa únicamente el segundo código.

CUARTA PRUEBA					
Ubicación Real		*Ubicación promedio obtenida con el algoritmo*		*Error*	
X	-0,98	X	-1,1311	0,1511	15,42%
Y	0,16	Y	0,0579	0,1021	63,81%
Z	-1,25	Z	-1,2418	0,0082	0,66%
Total Muestras	62			*Absoluto(m)*	*Relativo*

Tabla 12. Resultados prueba 4 slamnodistance.cpp

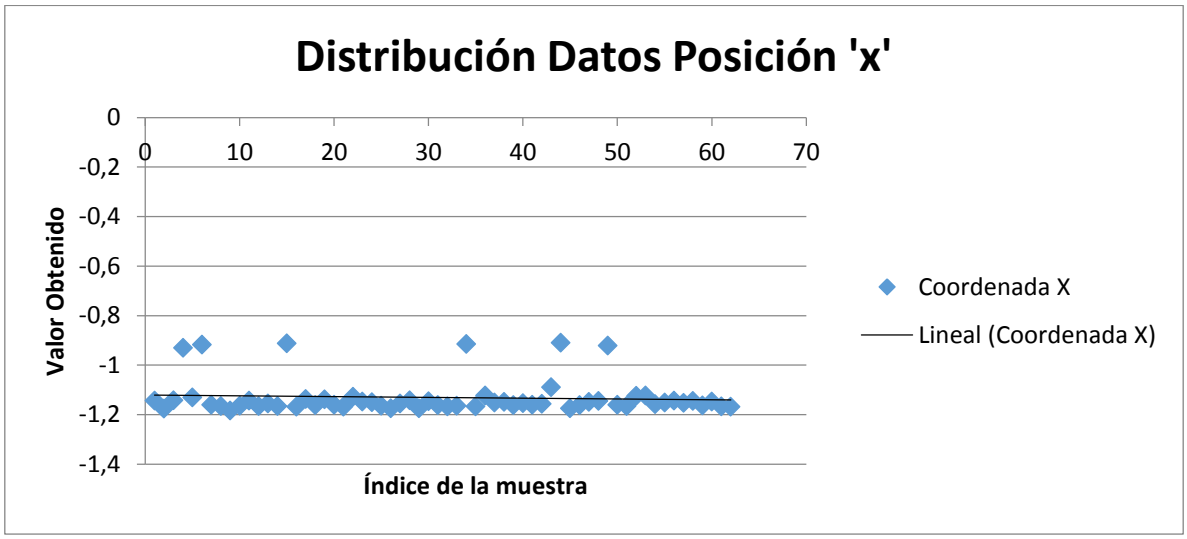


Figura 45. Distribución de datos posición 'x' para la cuarta prueba. Desviación estándar 0.072.

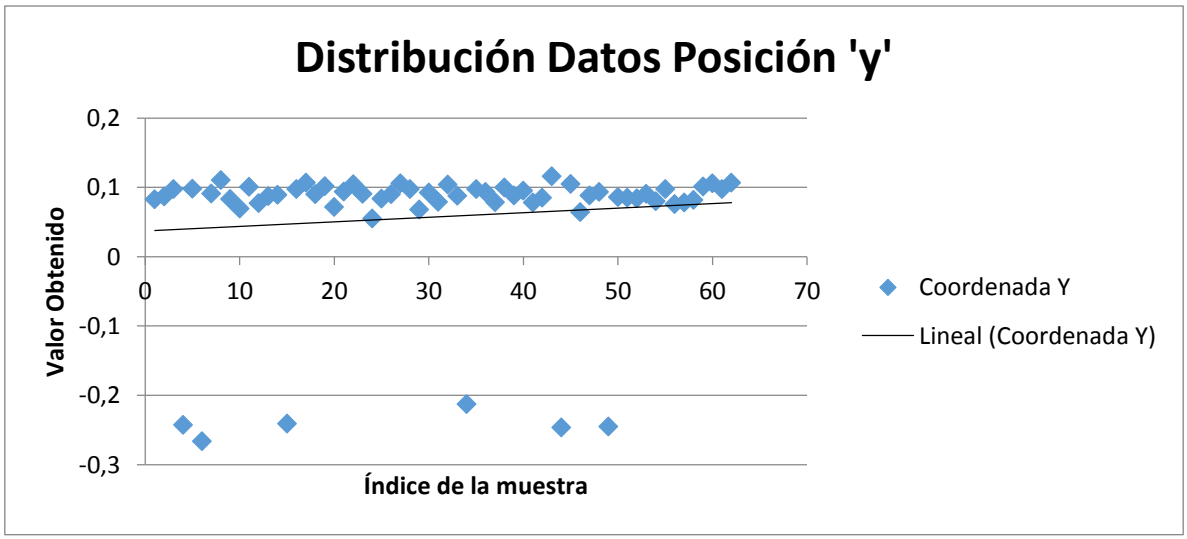


Figura 46. Distribución de datos posición 'y' para la cuarta prueba. Desviación estándar 0.099.

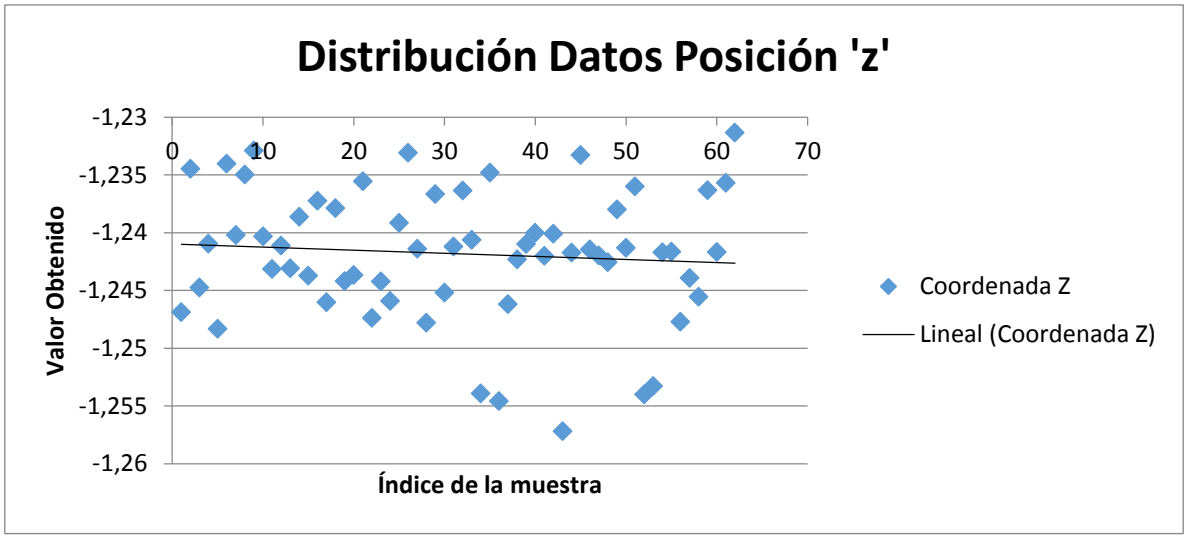


Figura 47. Distribución de datos posición 'z' para la cuarta prueba. Desviación estándar 0.005.

Como se puede observar, el error en la coordenada 'y' es demasiado alto y presenta de esta manera una incertidumbre muy alta, sin embargo esto es con respecto al error relativo, y este gran porcentaje se presenta debido a que el valor de distancia en 'y' es demasiado bajo. Si se observa en la tabla 12 el error absoluto en las coordenadas 'x' y 'y' es muy similar.

En general se puede afirmar que el error que presenta el algoritmo es relativamente bajo comparado con las distancias máximas que puede calcular.

En la figura 48 se observa la secuencia de códigos capturados por la cámara frontal del AR.drone en una ejecución del algoritmo slamnodistance.cpp. El orden de secuencia es de izquierda a derecha y de arriba hacia abajo.

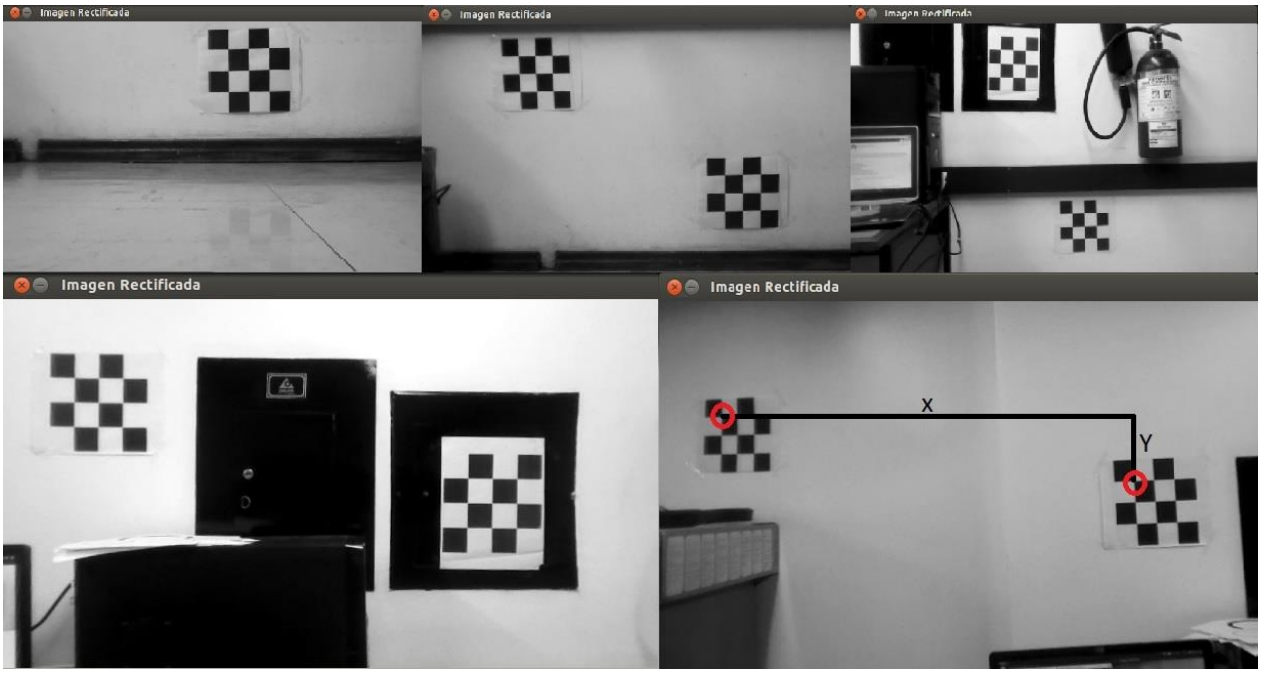


Figura 48. Secuencia de códigos identificados al ejecutar slamnodistance.cpp

Adicionalmente en la esquina inferior derecha de la figura 48 se observa la distancia en ‘x’ y ‘y’ que es calculada por el algoritmo entre los códigos identificados en ese frame.

5.2.5 ENTORNO DE SIMULACIÓN

Otra de las pruebas realizadas fue la identificación de códigos haciendo uso del entorno de simulación. En la figura 49 se puede observar el AR.drone dentro del entorno creado previamente y la forma en la cual se realiza la identificación de las esquinas de último código en la secuencia.

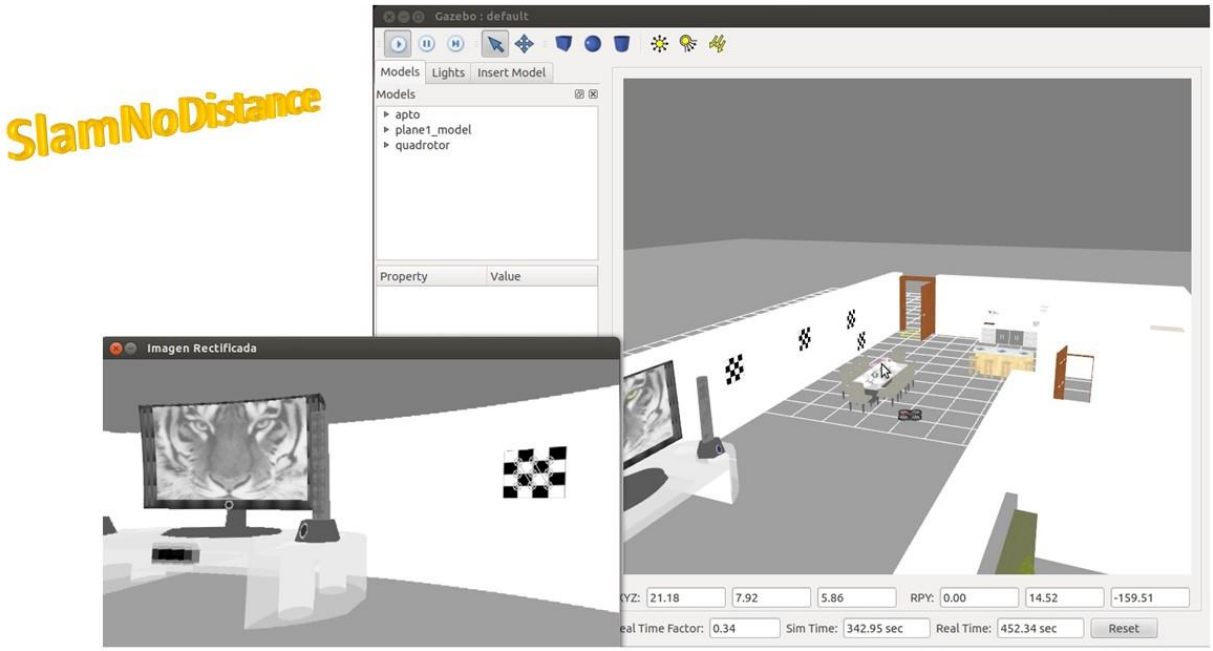


Figura 49. Prueba sobre entorno de simulación.

Bajo el entorno de simulación no tiene sentido analizar los resultados numéricos de posición dado que no es posible realizar una medición precisa de la posición real del AR.drone dentro del entorno de simulación para poder llevar a cabo una comparación de los resultados y de esta manera llegar a alguna conclusión basada en los estos.

5.3 TIEMPOS DE EJECUCIÓN

Con el fin de tener una información completa del desempeño de los algoritmos desarrollados se hicieron mediciones de tiempos de ejecución de diferentes partes del mismo y bajo condiciones diferentes.

En la tabla 13 se presentan los resultados obtenidos para el tiempo de captura de la imagen, es decir cuánto tarda la imagen en ser recibida en la estación base desde que es capturada por el AR.drone, este tiempo fue medido mediante herramientas propias del lenguaje de programación C, desde el momento en que se ejecuta algún algoritmo hasta cuando se tiene disponible la imagen lista para su procesamiento.

Tiempo de captura de imagen	
Tiempo Mínimo	0,05 s
Tiempo Máximo	0,07 s
Tiempo Promedio	0,06 s
Total Datos	40

Tabla 13. Tiempo de captura de la imagen en la estación base.

En la tabla 14 se muestran los resultados al medir varias veces el tiempo que tarda en ejecutarse los algoritmos cuando la imagen capturada por la cámara frontal del AR.drone no contiene códigos.

Tiempo de ejecución sin ver códigos	
Tiempo Mínimo	0,14 s
Tiempo Máximo	0,16 s
Tiempo Promedio	0,1496 s
Total Datos	45

Tabla 14. Tiempo de ejecución de los algoritmos cuando no se ven códigos en la imagen.

En la tabla 15 se muestran los resultados al medir varias veces el tiempo que tarda en ejecutarse los algoritmos cuando la imagen capturada por la cámara frontal del AR.drone contiene códigos.

Tiempo de ejecución viendo códigos	
Tiempo Mínimo	0,02 s
Tiempo Máximo	0,04 s
Tiempo Promedio	0,0265 s
Total Datos	43

Tabla 15. Tiempo de ejecución de los algoritmos cuando se ven códigos en la imagen.

Como se observa en las tablas 14 y 15 el algoritmo tarda más si la imagen capturada no contiene códigos, esto se debe a que el algoritmo intenta buscar las esquinas de los códigos durante más tiempo, lo cual prolonga el tiempo de ejecución alrededor de 100 ms.

En general el tiempo de captura de imágenes y procesamiento no supera los 200 ms, este es un tiempo suficiente para procesar una imagen por segundo, incluso podría pensarse en incrementar la cantidad de imágenes por segundo a procesar, sin embargo en pruebas realizadas se observó que si se incrementa el número de imágenes por segundo que tienen que ser procesadas, empiezan a hacerse evidentes las limitaciones de la conexión WiFi obteniendo un promedio de error mucho más elevado. Adicionalmente para esta aplicación, una imagen por segundo es una muy buena cantidad de información para ser mostrada en tiempo real.

6. CONCLUSIONES

En este punto vale la pena resaltar que los dos algoritmos presentan un desempeño bastante aceptable en el cálculo de las coordenadas del AR.drone en tiempo real, teniendo en cuenta siempre las restricciones presentadas por la iluminación y la estabilidad de la conexión WiFi ya que cuando la conexión se vuelve inestable se recibe una imagen distorsionada lo que implica pérdida de datos y fallos en los algoritmos.

Es importante evidenciar que cuando en las tablas que resumen los resultados de cada una de las pruebas se observan porcentajes de error del 20% o más, este valor hace referencia a un error relativo, y este error aumenta de manera dramática entre más pequeño sea el valor que se está midiendo, y que para el caso de nuestros algoritmos no es relevante tener un error relativo del 20% o 38% mientras el error absoluto se encuentra sobre los 6 cm.

Para el algoritmo slamnodistance.cpp hay que tener en cuenta que se presenta una acumulación de errores, dado que se presenta un error en el cálculo de la distancia entre los códigos y dicho error repercute de manera importante en el cálculo de las coordenadas 'x', 'y' y 'z' mostradas en tiempo real.

En general la coordenada 'z' es la que presenta un porcentaje de error más bajo a lo largo de todas las pruebas, esto debido a que el cálculo de los píxeles en los cuales se encuentran las esquinas de los tableros de ajedrez es mucho más preciso, ya que la forma en la cual se obtiene la coordenada 'z' incluye funciones de OpenCV mucho más robustas y que han tenido un proceso de evolución y corrección de errores bastante complejo.

El error absoluto más grande obtenido fue de 15 cm y el más pequeño de 0.1 mm. Teniendo en cuenta que en general el AR.drone durante un vuelo normal estando en estado estacionario, es decir 'quieto', se mueve alrededor de 5 cm, se puede afirmar que los rangos de error son completamente aceptables ya que este se verá afectado por las oscilaciones que presenta de manera natural el AR.drone.

La manera idónea mediante la cual se debe medir la efectividad de cada uno de los algoritmos durante un vuelo es usando un sistema muy preciso de cámaras que registran la posición del AR.drone en todo momento durante el vuelo y de esta manera poder comparar dato por dato lo obtenido por dicho sistema de cámaras con los datos que se obtienen mediante slammonocular.cpp y slamnodistance.cpp.

La estabilidad de la conexión WiFi y la iluminación juegan un papel determinante en el desempeño de los algoritmos desarrollados ya que si la conexión WiFi no es estable se presentan datos aleatoriamente que no coinciden con la medida promedio que está mostrando el algoritmo como se observó en algunas de las pruebas realizadas en este proyecto. Y por otro lado si la iluminación no es uniforme se presentan errores graves en la detección de los códigos lo que ocasiona una propagación de error evidente en los resultados finales.

Este trabajo de grado puede complementarse en un futuro, realizando un control para el AR.drone que tenga en cuenta la posición calculada en los algoritmos desarrollados y los datos de navegación, para que finalmente con base en éstos el robot posea una navegación completamente autónoma.

7. BIBLIOGRAFÍA

[1] Agarwal, S. (2012). Monocular Vision Based Indoor Simultaneous Localisation and Mapping for Quadrotor Platform.

[2] Wang, T.-C., & Chen, C.-H. (2013). Improved simultaneous localization and mapping by stereo camera and SURF. *2013 CACS International Automatic Control Conference (CACS)*, 204–209.

[3] George, L., & Mazel, A. (2013). Humanoid Robot Indoor Navigation Based on 2D Bar Codes : Application to the NAO Robot, *2013*.

[4] [En línea]. Portal fabricante de AR.drone 2.0. Disponible http://ardrone2.parrot.com/. [Último acceso: Noviembre 2014].

[5] George, E. A., Tiwari, G., Yadav, R. N., Peters, E., & Sadana, S. (2013). UAV systems for parameter identification in agriculture. *2013 IEEE Global Humanitarian Technology Conference: South Asia Satellite (GHTC-SAS)*, 270–273. doi:10.1109/GHTC-SAS.2013.6629929

[6] Kleiner, A., Prediger, J., & Nebel, B. (2006). RFID Technology-based Exploration and SLAM for Search And Rescue. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4054–4059. doi:10.1109/IROS.2006.281867

[7] Study, C., Sangin, N., & Johnson, C. W. (2006). Military Risk Assessment in Counter Insurgency Operations :, (June), 1–6.

[8] [En línea]. Wiki ROS. Documentación, tutoriales y soporte. Disponible http://www.ros.org/ [Último acceso: Noviembre 2014].

[9] [En línea]. Descarga y documentación del paquete desarrolado en ROS Ardrone_Autonomy. Disponible http://wiki.ros.org/ardrone_autonomy. [Último acceso: Noviembre 2014].

[10] [En línea]. Descarga y documentación del paquete desarrolado en ROS Tum_Ardrone. Disponible http://wiki.ros.org/tum_ardrone. [Último acceso: Noviembre 2014].

[11] [En línea]. Descarga y documentación del paquete desarrolado en ROS Tum_Simulator. Disponible http://wiki.ros.org/tum_simulator [Último acceso: Noviembre 2014].

[12] [En línea]. Portal OPENCV. Disponible http://opencv.org/ [Último acceso: Noviembre 2014].

[13] Bradski, G., Kaehler, A. (2008). Learning OpenCV. Computer Vision with the OpenCV Library. EEUU, O'Reilly Media, Inc. Páginas 370 – 378.

[14] Bradski, G., Kaehler, A. (2008). Learning OpenCV. Computer Vision with the OpenCV Library. EEUU, O'Reilly Media, Inc. Páginas 378 - 396.

[15] Bradski, G., Kaehler, A. (2008). Learning OpenCV. Computer Vision with the OpenCV Library. EEUU, O'Reilly Media, Inc. Páginas 381 - 384.

[16] Chatterjee, C., & Roychowdhury, V. P. (2000). Algorithms for coplanar camera calibration. Machine Vision and Applications, 12(2), 84–97. doi:10.1007/s001380050127

[17] Lee, S. J., Lim, J., Tewolde, G., & Kwon, J. (2014). Autonomous tour guide robot by using ultrasonic range sensors and QR code recognition in indoor environment. IEEE International Conference on Electro/Information Technology, 410–415. doi:10.1109/EIT.2014.6871799

[18] Raj, C. P. R., Tolety, S., & Immaculate, C. (2013). QR code based navigation system for closed building using smart phones. 2013 International Mutli-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s), 641–644. doi:10.1109/iMac4s.2013.6526488

8. ANEXOS

Dentro de los anexos de este trabajo de grado se encuentran los datos obtenidos en cada una de las pruebas realizadas, así como el código fuente de los dos algoritmos desarrollados y los archivos necesarios para poder ejecutar el entorno de simulación en cualquier computador que tenga instalado Gazebo.

Adicionalmente se podrán encontrar los seudocódigos de cada uno de los algoritmos desarrollados.