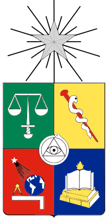


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE UNA LIBRERÍA DE CONTROL DE UN RADAR CON INTERFAZ A LA PLATAFORMA ROS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO
E INGENIERO CIVIL EN COMPUTACIÓN

CRISTOBAL GASPAR IGNACIO PIZARRO VENEGAS

PROFESORES GUÍA:
MARTIN ADAMS
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
DANIEL LÜHR SIERRA
JAVIER BUSTOS JIMENEZ

SANTIAGO DE CHILE
SEPTIEMBRE 2014

RESUMEN DE LA MEMORIA PARA OPTAR AL
TÍTULO DE INGENIERO CIVIL ELÉCTRICO
E INGENIERO CIVIL EN COMPUTACIÓN
POR: CRISTOBAL GASPAR IGNACIO PIZARRO VENEGAS
FECHA: SEPTIEMBRE 2014
PROFESORES GUÍA: MARTIN ADAMS Y NANCY HITSCHFELD

IMPLEMENTACIÓN DE UNA LIBRERÍA DE CONTROL DE UN RADAR CON INTERFAZ A LA PLATAFORMA ROS

Este trabajo de título se enmarca en la investigación realizada en el Centro de Tecnología Minera Avanzada (*Advanced Mining Technology Center*, AMTC) con tecnologías de percepción. Los desarrollos hechos en robótica en este centro constan de desarrollos de hardware y de software, siendo el software para robots un problema en si mismo. Actualmente, en el campo de la robótica, existe una plataforma de desarrollo que está tomando fuerza como una solución al problema del desarrollo de software para robots, la plataforma *Robot Operating System* (ROS), la cual permite el desarrollo de módulos de control para un robot como un sistema distribuido.

El objetivo de este trabajo de título es la implementación de una librería de control para un radar de ondas milimétricas que se usa en el AMTC, que permita su uso con la plataforma ROS. De esta forma, el radar podrá ser usado como un sensor en conjunto con las otras piezas de hardware (sensores y actuadores) de que dispone el laboratorio.

Esta librería consiste en un conjunto de piezas de software ejecutables que permiten la comunicación con el radar, generando datos en el formato apropiado para la plataforma, una interfaz gráfica que facilita el control del sistema, un módulo que permite la visualización de datos y un módulo que implementa un algoritmo de detección de objetos basado en los datos generados por los otros módulos.

La librería implementada fue probada en el laboratorio, en un entorno controlado, para la validación directa de ésta contra el software oficial que se vende con el radar, y para analizar el uso de la red por parte de la librería. Además, se probó en la elipse del parque O'Higgins, para un caso de prueba en terreno, en la que se evaluó el desempeño de la integración del radar con ROS, usando sus funcionalidades de visualización de datos.

Con este trabajo se logró la utilización de todas las características del radar y su total integración con la plataforma ROS, lo que permitirá el uso del radar con los otros sensores del laboratorio y su uso para nuevos experimentos.

A mi padre por ponerme en un camino que disfruto y seguiré disfrutando.
A mi madre por apoyarme en los múltiples detallitos de la vida de escolar y universitario.

Agradecimientos

A mis profesores guía Martin Adams y Nancy Hitschfeld, por la disposición mostrada para esta memoria, considerando lo peculiar que es hacerla para las dos carreras en las que estoy involucrado.

A Daniel Lühr, por el tiempo dado para esta memoria, su disposición a resolver dudas, su ayuda en lo práctico y en lo teórico, y por acompañarme a hacer los experimentos, ya que es demasiado vergonzoso manejar un robot desde la Facultad hasta el parque O'Higgins solo.

Tabla de contenido

Resumen I

Agradecimientos III

1. Introducción 3
- 1.1. Motivación . . . 3
- 1.2. Objetivos del proyecto . . . 4
  - 1.2.1. Objetivo general . . . 4
  - 1.2.2. Objetivos específicos . . . 4
- 1.3. Estructura de la memoria . . . 5

2. Antecedentes 6
- 2.1. Radar . . . 6
  - 2.1.1. Medición de distancias con radar . . . 7
    - 2.1.1.1. Tiempo de vuelo (Time-of-flight, TOF) . . . 7
    - 2.1.1.2. Onda continua modulada en frecuencia (FMCW) . . . 8
  - 2.1.2. Medidas en un radar y pre-procesamiento . . . 10
    - 2.1.2.1. Compresión logarítmica . . . 10
    - 2.1.2.2. Compensación de rango . . . 11
  - 2.1.3. Post-procesamiento de mediciones . . . 11
    - 2.1.3.1. Aplicación de umbral fijo . . . 11
    - 2.1.3.2. Aplicación de umbral adaptativo . . . 12
- 2.2. Robot Operating System . . . 13
  - 2.2.1. Servidor de parámetros . . . 14
  - 2.2.2. Roslaunch . . . 15
  - 2.2.3. Rviz y Rqt_plot . . . 15
  - 2.2.4. Rosbag . . . 15
- 2.3. Equipos utilizados . . . 16
  - 2.3.1. Radar . . . 16
  - 2.3.2. Láser . . . 18
  - 2.3.3. Base móvil . . . 19
- 2.4. Experiencias similares y el aporte de esta memoria . . . 19

3. Diseño e implementación 20
- 3.1. Requerimientos . . . 20
- 3.2. Decisiones de diseño . . . 21

3.2.1. Lenguajes de programación 21
3.2.2. Interfaz a ROS 21
3.2.3. Interfaz gráfica 22
3.2.4. Visualización de datos 22
3.3. Estructura general del software 22
3.4. Decisiones incidentales 24
3.4.1. Lectura y escritura de mensajes 24
3.4.2. *Sockets* y sincronización 24
3.4.3. Robustez de la conexión 25
3.4.4. Parámetros del radar y configuración 25
3.4.5. Medición de datos desde el radar 25
3.5. Funcionalidades implementadas 26
3.5.1. Nodos de comunicaciones 26
3.5.2. Interfaz gráfica 27
3.5.3. Nubes de puntos 28
3.5.4. Detección de obstáculos 29
3.5.5. Lanzadores 29
3.6. Modos de uso del sistema 30

4. Resultados **31**
4.1. Desempeño en la red 31
4.2. Pruebas de *scanner* 32
4.3. Pruebas de radar 35
4.4. Prueba en terreno 37
4.4.1. Nubes de puntos 39
4.4.2. Detección de obstáculos 40
4.4.3. Comparación de fuentes de datos 44

5. Conclusiones **47**
5.1. Trabajos futuros 48

Bibliografía **50**

Anexos **51**
A. Interfaz a ROS del cliente implementado 51
B. Diferencias encontradas entre [1] y el verdadero protocolo del radar 53

Índice de tablas

2.1. Especificaciones técnicas del radar utilizado. 18
2.2. Especificaciones técnicas del sensor láser utilizado. 18
2.3. Especificaciones técnicas de la base móvil utilizada. 19

4.1. Ancho de banda ocupado por cada servidor en distintos modos de operación del radar. 31

Índice de figuras

2.1. Principio de funcionamiento del radar. Al cambiar de medio, una onda emitida (línea negra), se divide en dos: Una onda reflejada (línea roja) y una onda transmitida (línea gris). 6
2.2. Mediciones tomadas por un radar en una habitación rectangular. Se puede ver que los máximos en cada medición angular forman un rectángulo, mostrando las paredes del entorno. Tomado de [2]. 7
2.3. Diagrama de tiempo de la técnica de tiempo de vuelo. El tiempo transcurre de arriba hacia abajo y las distancias se miden de izquierda a derecha. Primero, el radar envía un pulso, el cual se refleja y transmite en el obstáculo 1. Luego se refleja en el obstáculo 2. Finalmente, el radar recibe los pulsos reflejados, con los que se calcula la distancia usando la ecuación 2.1. 8
2.4. Ondas generadas en un radar FMCW. El gráfico superior muestra la señal que emite el radar, y el gráfico inferior muestra la frecuencia de esta señal. En este caso la frecuencia de la onda portadora es de 10 Hz. 9
2.5. Interacción entre las variables asociadas en la ecuación 2.2. 9
2.6. Radar Acumine. 16
2.7. Interfaz lógica ofrecida por el radar [1]. 17
2.8. Estructura general de los mensajes enviados por el radar. 18
2.9. Sensor láser SICK. 18
2.10. Base móvil Husky. 19

3.1. Estructura general de los nodos implementados y su interfaz a ROS. 23
3.2. Interfaz gráfica del sistema implementado. 27

4.1. Control de rapidez del *scanner*. 32
4.2. Primera prueba de control de posición del *scanner*. 33
4.3. Segunda prueba de control de posición del *scanner*. 34
4.4. Tercera prueba de control de posición del *scanner*. 34
4.5. Modo de *peak* más potente. 35
4.6. Modo de *peak* más cercano. 36
4.7. Modo de *peak* más lejano. 36
4.8. Modo de todos los *peaks*. 36
4.9. Modo de espectro en frecuencia. 37
4.10. Ubicación del lugar de pruebas. Los puntos en rojo muestran los lugares donde se ubicó el radar para hacer las mediciones. Fuente: Google Maps. 38
4.11. Configuración del sistema utilizada en las pruebas. 38
4.12. Modo de *peak* más potente. 40

4.13. Modo de *peak* más cercano. 40
4.14. Modo de *peak* más lejano. 40
4.15. Modo de todos los *peaks*. 40
4.16. Vista de espectro en frecuencia. Las medidas en negro y blanco son de bajas y altas potencias recibidas respectivamente. 41
4.17. Obstáculos detectados por el procesador CFAR tridimensional. Las medidas en rojo y blanco son las detecciones de menor y mayor P_D^{CFAR} respectivamente. 42
4.18. Obstáculos detectados por el procesador CFAR unidimensional. Las medidas en rojo y blanco son las detecciones de menor y mayor P_D^{CFAR} respectivamente. 43
4.19. Medidas obtenidas por el sensor láser en conjunto con las medidas del radar. Las primeras son de color rojo, mientras que las segundas son en tonos de blanco y negro. 45
4.20. Mediciones unificadas entre dos puntos. Los tonos de blanco y negro corresponden a bajas y altas intensidades respectivamente. Los círculos en negro corresponden a cruces entre dos pares de reflexiones múltiples. Comparado con la Figura 4.16, en esta Figura la cámara se ubica a la derecha y mira hacia la izquierda. 46

Glosario

CA-CFAR: *Cell Averaging CFAR*, CFAR de Promedio de Celdas.

CFAR: *Constant False Alarm Rate*, Tasa Constante de Falsa Alarma.

FLANN: *Fast Library for Approximate Nearest Neighbors*, Librería Rápida de Vecinos más Cercanos Aproximados.

FMCW: *Frequency Modulated Continuous Wave*. Onda Continua Modulada en Frecuencia.

LASER: *Light Amplification by Stimulated Emission of Radiation*, Amplificación de Luz mediante Emisión Estimulada de Radiación.

PCL: *Point Cloud Library*, Librería de Nubes de Puntos.

PID: *Proportional Integral Derivative*. Proporcional Integral Derivativo.

RADAR: *Radio Detection and Ranging*, Detección y Medición de Distancias con Radio.

RCS: *Radar Cross Section*, Sección Transversal de Radar.

ROS: *Robot Operating System*, Sistema Operativo de Robots.

TCP: *Transmission Control Protocol*, Protocolo de Control de Transmisión.

TOF: *Time Of Flight*, Tiempo de Vuelo.

XML: *eXtensible Markup Language*, Lenguaje de Marcado Extensible.

YAML: *Yet Another Markup Language*, Otro Lenguaje de Marcado.

Capítulo 1

Introducción

Actualmente, en el ámbito de la robótica móvil, un problema importante es la medición de distancias, con las que un robot puede percibir su entorno. Por otro lado, la medición de distancias es un problema en si mismo, que no tiene que estar asociado a la robótica móvil, como en el caso de generación de mapas digitales de terreno. Para resolver el problema de medición de distancias se dispone de diversas formas y técnicas, entre ellas sensores láser, cámaras estereoscópicas, sonares y, la más relevante para este trabajo, el radar. Estos sensores tienen diversas ventajas y desventajas, siendo para ciertas aplicaciones unos más apropiados que otros, no habiendo una solución única o un sensor definitivo para todos los problemas de medición de distancias. Por ejemplo, el radar suele ser un sensor de precisión inferior al láser, pero da más información acerca del entorno que el láser y puede funcionar en condiciones en que el láser no funciona, como en ambientes polvorientos o con lluvia, de forma que se puede aprovechar esta información para complementar las mediciones del láser y combinar la información de ambos sensores.

1.1. Motivación

Esta memoria de ingeniería se enmarca en el ámbito de la investigación en robótica, particularmente en la percepción espacial de un robot móvil, que se aplica en problemas de localización, teleoperación y construcción de mapas para un robot. En el Centro Avanzado de Tecnología para la Minería (AMTC) se desarrolla investigación en los campos de la ingeniería con aplicaciones a la minería, como son la exploración y modelamiento de yacimientos y la automatización en minería, entre otras [3]. En el campo de automatización en minería se desarrollan aplicaciones robóticas, que usan robots móviles. En particular, en el Laboratorio de Robótica de Campo, se tiene un robot móvil equipable con diversos sensores, como un

sensor láser, un sensor Kinect[1], una cámara estéreo[2] y un radar. Este radar no es compatible con los otros sensores, ya que el software que lo controla y le permite capturar datos genera datos en un formato que solo sirve para almacenarlos, no permitiendo un análisis en tiempo real ni la interoperabilidad directa con los otros sensores. Se han hecho esfuerzos por integrar los componentes de este robot en la plataforma *Robot Operating System* (ROS) [4], para acelerar el desarrollo de nuevas funcionalidades para la investigación que se pueda hacer con el robot. Sin embargo, queda la integración del radar a ROS, de forma que se pueda manejar el radar de forma estándar con el robot y además se puedan usar los desarrollos abiertos proporcionados por la comunidad de ROS.

1.2. Objetivos del proyecto

1.2.1. Objetivo general

El objetivo general de esta memoria es la implementación de una completa librería de control de un radar con interfaz sobre la plataforma ROS. Esta librería debe poder permitir la utilización de los datos generados por el radar en la plataforma y la configuración precisa de éste.

1.2.2. Objetivos específicos

1. Implementar los módulos de ROS en una interfaz de línea de comandos que permitan realizar lo siguiente:
    a) Controlar la posición y rapidez angular del espejo del radar.
    b) Controlar la resolución y el modo de procesamiento de la potencia recibida por el radar.
    c) Integrar múltiples lecturas angulares en una observación completa de 360°.
2. Implementar una interfaz gráfica de usuario, que permita acceder a las funcionalidades de los módulos ya implementados.
3. Implementar un algoritmo de detección de objetos usando los módulos implementados.

[1]Este es un sensor compuesto por un par de cámaras y un sensor de profundidad, desarrollado por Microsoft, que permite obtener imágenes con profundidad del entorno.

[2]Estos sensores funcionan con dos cámaras y permiten estimar profundidades de la misma forma que hacen los seres humanos.

1.3. Estructura de la memoria

La estructura de esta memoria de título es la siguiente:

Antecedentes: En este capítulo se describen los conceptos necesarios para comprender esta memoria, generalidades acerca de las tecnologías de radar, sobre procesamiento de datos, sobre la plataforma ROS y sobre el radar que se usa en este trabajo.

Diseño e implementación: En este capítulo se describe el software implementado, las funcionalidades logradas, las decisiones de diseño tomadas en la implementación del programa y los modos de uso de éste.

Resultados: En este capítulo se muestran las pruebas realizadas con el radar que muestran el correcto funcionamiento del software implementado y las funcionalidades que se pueden obtener con el radar.

Conclusiones: En este capítulo se muestra lo que se logró con esta memoria y desarrollos futuros posibles.

Capítulo 2

Antecedentes

En este capítulo se describen las tecnologías usadas para esta memoria, a nivel de hardware, correspondiente al radar y sus características, como a nivel de software, esto es la plataforma ROS y sus funcionalidades ofrecidas para el desarrollo.

2.1. Radar

RADAR, acrónimo de *Radio Detection and Ranging*[1] es una tecnología de percepción basada en la emisión de ondas electromagnéticas, específicamente ondas de radio (30 MHz - 300 GHz) que permite la detección de obstáculos, además de medición de distancias y velocidades.

El principio en el que se basa el radar es que cuando una onda pasa por un medio que cambia de densidad, por ejemplo aire y madera, una porción de la onda pasa al otro lado y la otra se refleja, como muestra la Figura 2.1.

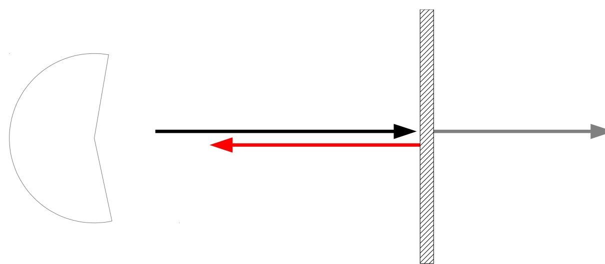


Figura 2.1: Principio de funcionamiento del radar. Al cambiar de medio, una onda emitida (línea negra), se divide en dos: Una onda reflejada (línea roja) y una onda transmitida (línea gris).

Las ondas reflejadas son recibidas por el radar y esto permite obtener información sobre

[1]A pesar de ser un acrónimo, RADAR será utilizado como una palabra cualquiera de aquí en adelante.

la línea que ha recorrido la onda y, previo procesamiento de los datos, se puede obtener un conjunto de medidas de potencia recibida v/s distancia, que indican si a cierta distancia está o no un objeto, como muestra la Figura 2.2.

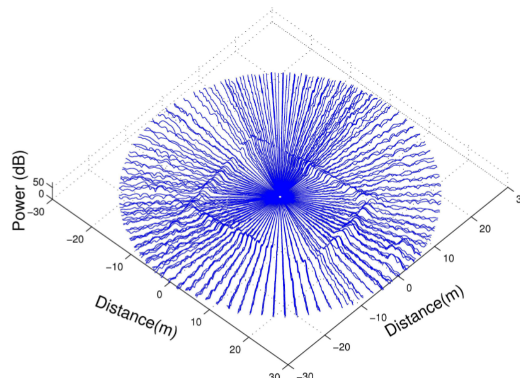


Figura 2.2: Mediciones tomadas por un radar en una habitación rectangular. Se puede ver que los máximos en cada medición angular forman un rectángulo, mostrando las paredes del entorno. Tomado de [2].

Los radares actualmente tienen aplicaciones civiles y comerciales en los campos de:

- Minería
- Vehículos inteligentes
- Localización
- Navegación marítima

entre otros [5, sección 1.3].

2.1.1. Medición de distancias con radar

Las dos técnicas de medición de distancias con radar más populares son las siguientes:

2.1.1.1. Tiempo de vuelo (Time-of-flight, TOF)

Esta técnica consiste en la emisión de pulsos electromagnéticos desde el radar, y la distancia se mide proporcional al tiempo entre el envío y recepción de éstos, con la ecuación 2.1.

$$r_i = \frac{cT_i}{2} \tag{2.1}$$

donde r_i es la distancia entre el radar y el objeto i, c es la velocidad de la luz y T_i es el tiempo entre que se envía un pulso y el radar recibe un eco. El diagrama de tiempos que muestra el

funcionamiento de la técnica se muestra en la Figura 2.3.

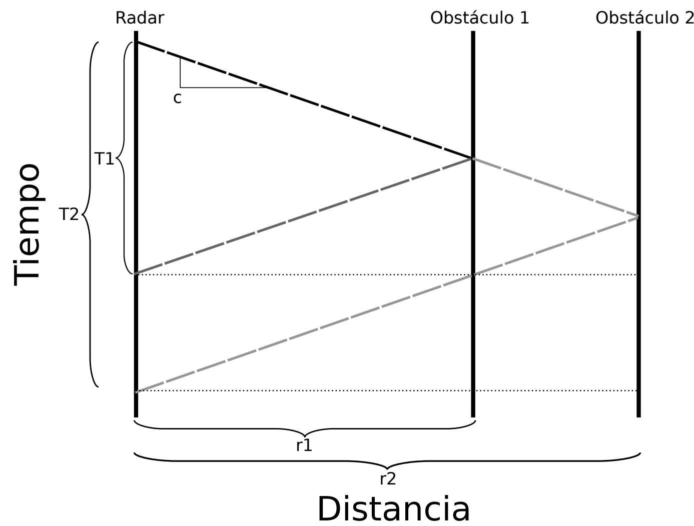


Figura 2.3: Diagrama de tiempo de la técnica de tiempo de vuelo. El tiempo transcurre de arriba hacia abajo y las distancias se miden de izquierda a derecha. Primero, el radar envía un pulso, el cual se refleja y transmite en el obstáculo 1. Luego se refleja en el obstáculo 2. Finalmente, el radar recibe los pulsos reflejados, con los que se calcula la distancia usando la ecuación 2.1.

El principal problema de esta técnica es que, al ser necesario enviar pulsos, la electrónica de transmisión y recepción del radar tiene que poder manejar grandes potencias, del orden de los 4 kW, y poder procesar las señales enviadas a altas velocidades. Además, con este método se tiene una precisión que no sirve para aplicaciones de localización [5, sección 2.6.1].

2.1.1.2. Onda continua modulada en frecuencia (FMCW)

Esta técnica consiste en la emisión continua de una onda, con una frecuencia portadora f_c, la que es modulada por una curva de dientes de sierra, y la distancia se mide de forma inversamente proporcional a la diferencia entre la frecuencia de la onda enviada y la frecuencia de la onda recibida. Las señales modulada y de frecuencia se muestran en la Figura 2.4.

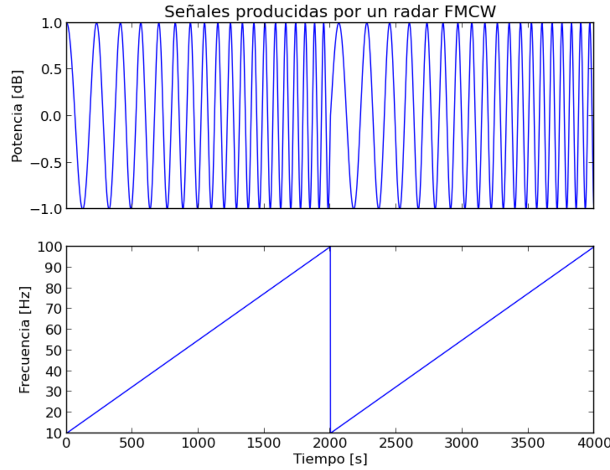


Figura 2.4: Ondas generadas en un radar FMCW. El gráfico superior muestra la señal que emite el radar, y el gráfico inferior muestra la frecuencia de esta señal. En este caso la frecuencia de la onda portadora es de 10 Hz.

Entonces la distancia se mide con la ecuación 2.2.

$$r_i = \frac{cT_d}{2\Delta f} f_{bi} \tag{2.2}$$

donde r_i es la distancia entre el radar y el objeto i, Δf es la amplitud del barrido de frecuencias del radar, T_d es el periodo de la curva de frecuencia y f_{bi} es la diferencia entre la frecuencia recibida por el radar proveniente del objeto i y la frecuencia portadora f_c. La interacción entre estas variables se muestra en la Figura 2.5.

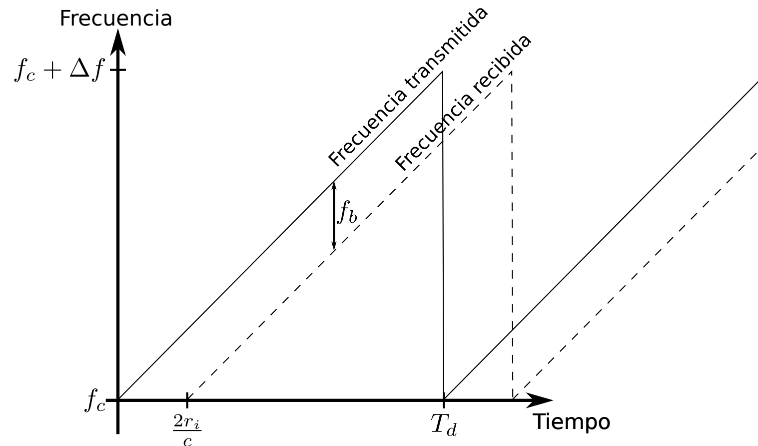


Figura 2.5: Interacción entre las variables asociadas en la ecuación 2.2.

2.1.2. Medidas en un radar y pre-procesamiento

Con los métodos referidos anteriormente en 2.1.1 es posible obtener, para una dirección determinada, un conjunto de medidas de potencia recibida, llamada un *A-scope*, que se denota de la siguiente manera:

$$S^{lin} = [S_1^{lin}, S_2^{lin}, \ldots, S_Q^{lin}] \tag{2.3}$$

Con esto, el rango del radar se discretiza en Q secciones o *bins* y para cada una de ellas se tiene la potencia recibida en esa sección. Esta potencia se mide en Watts, pero usualmente S^{lin} es una medición sin unidades, ya que las medidas son relativas a la potencia transmitida o cualquier otra potencia [5, sección 2.2]. Por otro lado, cuando hay un obstáculo, la potencia recibida por un radar decae con la cuarta potencia de la distancia a la que está éste, es decir

$$S_i^{lin} \propto \frac{\Gamma^{RCS}}{r_i^4} \tag{2.4}$$

$$S_i^{lin} = K\frac{\Gamma^{RCS}}{r_i^4} \tag{2.5}$$

Donde S_i^{lin} es la medición del radar en la sección i y Γ^{RCS} es llamada Sección Transversal de Radar, o *Radar Cross Section* (RCS), una propiedad que encapsula las propiedades físicas del obstáculo, siendo algo como la "brillantez" de éste ante el radar. Un problema con esto es que la RCS puede variar con rangos muy grandes dependiendo de la distancia a la que está un obstáculo y el ángulo desde el cual se ve, lo que hace necesario un preprocesamiento de los datos recibidos por el radar para que estos puedan ser procesados por un computador.

2.1.2.1. Compresión logarítmica

Se toma el logaritmo de la señal recibida por el radar (ecuación 2.5), obteniéndose lo siguiente:

$$S_i^{log} = 10log_{10}\left(S^{lin}\right) = 10log_{10}\left(K\frac{\Gamma^{RCS}}{r_i^4}\right) \tag{2.6}$$

$$S_i^{log} = 10log_{10}(K) + 10log_{10}\left(\Gamma^{RCS}\right) - 40log_{10}(r_i) \tag{2.7}$$

Con esto la dependencia del logaritmo de la señal recibida se vuelve lineal con el logaritmo de la distancia. Nótese que a partir de este punto las medidas dejan de ser en Watts (o adimensionales, si se usan potencias relativas), sino que son en decibeles.

2.1.2.2. Compensación de rango

En los radares FMCW se tiene que la distancia de un obstáculo es proporcional a la frecuencia recibida por el radar, es decir

$$f_{bi} = K_f r_i \tag{2.8}$$

Entonces, se puede aplicar un filtro pasa alto respecto a la frecuencia, con lo que de la ecuación 2.7 se tiene:

$$\begin{aligned}
S_i^{log_rc} &= S_i^{log} + 40log_{10}(f_{bi}) \qquad [40 \text{ dB/década}] \\
S_i^{log_rc} &= 10log_{10}(K) + 10log_{10}(\Gamma^{RCS}) - \cancel{40log_{10}(r_i)} + 40log_{10}(K_f) + \cancel{40log_{10}(r_i)} \\
S_i^{log_rc} &= K^* + 10log_{10}(\Gamma^{RCS})
\end{aligned} \tag{2.9}$$

con K^* una constante. Entonces la señal recibida se vuelve independiente de la distancia a la que esté el obstáculo, quedando solo dependiente de la RCS.

Los preprocesamientos descritos hacen, finalmente, que blancos iguales sean vistos de igual forma, independientemente de la distancia hacia el radar. Un problema con estos métodos de preprocesamiento es que funcionan cuando las mediciones corresponden a blancos reales, ya que todas las ecuaciones anteriores involucran la RCS en Γ^{RCS}. Sin embargo, para las mediciones que no corresponden a obstáculos, sino que solamente corresponden a ruido, los métodos de preprocesamiento resultan en una señal cuyo valor aumenta con la distancia, lo cual puede causar problemas para la detección de obstáculos.

2.1.3. Post-procesamiento de mediciones

Después de hacer el preprocesamiento de las mediciones como se describe en la sección 2.1.2, se tiene una curva que se puede interpretar como la potencia "vista" por el radar en función de la distancia. Lo que se debe hacer con esta señal es decidir, para cada punto de ésta, si lo que se está viendo es un obstáculo o solo ruido. Para ello se puede hacer lo siguiente:

2.1.3.1. Aplicación de umbral fijo

Una forma directa para detectar obstáculos es establecer un umbral, y asumir que las mediciones que superan ese umbral son obstáculos y las mediciones bajo éste son ruido. A pesar de ser simple de implementar, esta técnica necesita definir el umbral a priori, lo que puede causar que para cierto umbral ciertas mediciones sean detectadas como obstáculos mientras que con otro umbral sean descartadas como ruido, no habiendo una forma intuitiva o matemáticamente justificada para elegir el umbral. Además, al no modelar la señal probabilísticamente, no se puede obtener información acerca de los valores mismos de la señal, es

decir, no se puede saber qué diferencia hay entre dos puntos que superan el umbral establecido, y no toman en cuenta la incertidumbre respecto a la RCS, con lo que éste método es propenso a fallas [5, sección 3.3.1].

2.1.3.2. Aplicación de umbral adaptativo

Este enfoque asume que cada medida del radar es una muestra de una variable aleatoria cuya distribución de probabilidad depende de la presencia o ausencia de objetos en el rango que se está midiendo, esto es, para cada medición S se asumen dos hipótesis:

- S proviene del ruido ambiental (H_0). En este caso, se asume que el ruido tiene una distribución exponencial, esto es:

$$P(S|H_0) = \begin{cases} \lambda \exp(-\lambda S) & \text{si } S \geq 0 \\ 0 & \text{si } S < 0 \end{cases} \tag{2.10}$$

- S proviene de un obstáculo (H_1). En este caso, se asume que la medición sigue una distribución de Rayleigh, esto es:

$$P(S|H_1) = \frac{x}{\sigma^2} \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{2.11}$$

Con estas dos distribuciones, se puede establecer un umbral que garantice una cierta probabilidad de falsa alarma (Constant False Alarm Rate, CFAR), esto es que el sistema decida que hay un obstáculo cuando solo se está leyendo ruido, P_{FA}^{CFAR}, dada por

$$S^{CFAR} = -\lambda^{-1} \ln\left(P_{FA}^{CFAR}\right) \tag{2.12}$$

El problema que este método presenta al ser aplicado directamente es que, al igual que la aplicación de umbral fijo, necesita parámetros definidos a priori (λ y σ). La forma de manejar este problema es haciendo una estimación adaptativa de los parámetros, usando los mismos datos. En términos generales, estos algoritmos (conocidos como *procesadores*) consisten en que para una medición S_i se aplica un test estadístico T, y si la razón $\frac{S_i}{T(S_i)}$ supera un umbral γ, se dice que la medición corresponde a un obstáculo. Diferentes modelos probabilísticos de obstáculos y ruido, a la vez que diferentes esquemas de procesamiento, definen distintos T y γ. Un caso es el procesador de promedio de celdas (*Cell Averaging* CFAR, CA-CFAR). Este procesador tiene los siguientes parámetros:

- Una probabilidad de falsa alarma P_{FA}^{CFAR}.
- Un ancho de ventana de referencia W.
- Un ancho de ventana de guardia G.

Entonces, para cada medición S_i el test T se define como

$$T(S_i) = \frac{1}{K} \sum_{k=1}^{k=K} S_{i_k} \tag{2.13}$$

Donde $\{S_{i_1}, S_{i_2}, \cdots, S_{i_k}\}$ son las mediciones que quedan espacialmente en el anillo definido por G y W en torno a S_i. Con distribuciones de probabilidad para ruido y obstáculos como en las ecuaciones 2.10 y 2.11 respectivamente, el umbral γ es

$$\gamma = K \left[\left(P_{FA}^{CFAR} \right)^{\frac{-1}{K}} - 1 \right] \tag{2.14}$$

y la probabilidad de detección se estima como

$$P_D^{CFAR}(S_i) = \left[1 + \frac{\gamma}{K(1 + \frac{S_i}{T(S_i)})} \right]^{-K} \tag{2.15}$$

De esta forma, el procesador CA-CFAR (referido en adelante solamente como CFAR), hace que si una medida es alta, pero está rodeada de otras medidas altas, se tome como ruido, pero si las medidas que la rodean son bajas, se detecte como un obstáculo.

2.2. Robot Operating System

Robot Operating System (ROS) es un meta-sistema operativo desarrollado por la empresa Willow Garage[2] para el desarrollo de software para robots, que ofrece comunicación entre procesos, *drivers* para distintos sensores y actuadores, y un sistema de compilación, construcción y manejo de paquetes[3]. Actualmente esta plataforma corre sobre sistemas Unix, principalmente sobre Ubuntu y Mac OS, y, aunque es posible, no se ha desarrollado completamente una adaptación a Windows[4, Sección 3].

ROS es un sistema que ha sido pensado para la reutilización de código[4, Sección 2], esto es que, por ejemplo, un algoritmo de reconocimiento de imágenes pueda ser portado fácilmente de un robot a otro, al organizar los programas hechos en *paquetes*, que pueden funcionar independientemente, o incluir dependencias de otros paquetes. Por otro lado, existe una comunidad abierta de desarrollos en torno a ROS[4], en la que se pueden encontrar paquetes para diversas funcionalidades.

ROS es un sistema distribuido, cuyos programas se comunican mediante la red, usando un protocolo propio del sistema, de especificación abierta [6], por lo tanto éstos pueden ser

[2] `http://willowgarage.com`

[3] Es por esto que se le llama un meta-sistema operativo a ROS, porque las funciones descritas son propias de un sistema operativo, sin embargo ROS es un sistema que debe correr sobre un sistema operativo normal.

[4] `http://www.ros.org/browse/list.php`

programados en cualquier lenguaje. Actualmente existen librerías clientes que permiten la programación en lenguajes C++, Python y Lisp, mientras que se tienen librerías experimentales para programar en lenguajes Java y Lua. Esto permite que en un sistema o paquete, por ejemplo, secciones de alto desempeño puedan ser programadas en un lenguaje eficiente como C++, mientras que interfaces gráficas u operaciones de alto nivel puedan ser programadas en un lenguaje más simple y rico en librerías como Python.

Algunos conceptos básicos sobre el funcionamiento de ROS se muestran a continuación.

Nodo: Un nodo es un programa ejecutable en ROS, que puede estar dedicado a una función específica, como leer de un sensor, procesar datos, etc. Estos nodos pueden comunicarse con otros con las herramientas que ofrece el sistema.

Mensaje: Un mensaje es un dato, con una estructura inamovible y rigurosamente definida, que puede transmitirse entre los nodos de ROS. Estos mensajes se declaran en archivos de texto simple, en un lenguaje de dominio específico para estos mensajes, fuera del código de los nodos, mientras que las herramientas de compilación generan las estructuras de datos o clases para poder ser usados en los nodos. Así se posibilita la comunicación de mensajes entre nodos escritos en distintos lenguajes.

Tópico: Junto con los servicios, los tópicos son una de las formas estándares de comunicación entre nodos de ROS, que permite que los nodos se comuniquen con mensajes, descritos anteriormente. Esta forma de comunicación se basa en el patrón de publicadores y suscriptores[7], que permiten que un nodo pueda suscribirse, es decir obtener mensajes, de varios tópicos, y que varios nodos puedan publicar en el mismo tópico, es decir que un nodo escuchando un tópico está recibiendo mensaje de más de un nodo.

Servicio: Un servicio es una forma de comunicación síncrona entre nodos, que se comunican usando pares de mensajes, un mensaje con los datos de petición de servicio (por ejemplo, dos enteros), y otro con el dato de respuesta del servicio, (por ejemplo, si el servicio es sumar dos enteros, el tipo de dato de respuesta sería un entero). Esta forma de comunicación es uno a uno, esto es que la respuesta a un servicio se entrega a un solo nodo (aunque un servicio puede ser llamado por distintos nodos), al contrario de los tópicos, y permite una comunicación transparente entre nodos remotos.

ROS también ofrece herramientas para el desarrollo y despliegue de aplicaciones, que se muestran a continuación.

2.2.1. Servidor de parámetros

ROS provee un servicio para guardar parámetros, que permite que distintos nodos puedan compartir información, y que se pueda tener un estado común para un sistema[5]. Este servidor

[5]Con sistema, de aquí en adelante, se hace referencia a un conjunto de nodos corriendo simultáneamente, ya sean de un solo paquete, o nodos de distintos paquetes.

puede ser accedido mediante el programa `rosparam` en línea de comandos, o con las librerías clientes, como las de Python y C++. Con esto, un nodo puede definir un parámetro en el servidor de parámetros, y así éste se vuelve disponible para todos los demás nodos.

2.2.2. Roslaunch

Roslaunch es una herramienta de ROS que permite la ejecución de múltiples nodos con una sola llamada, a la vez que definir parámetros y reiniciar procesos terminados. Roslaunch usa archivos XML o YAML (lanzadores), de extensión `.launch` para definir nodos a correr, parámetros para cargar en el servidor de parámetros, e incluso otros archivos `.launch` para correr. Esto permite hacer que en un sistema compuesto por muchos nodos que ofrecen distintas funcionalidades, se puedan tener archivos `.launch` que permitan correr fácilmente distintas funcionalidades del sistema, por ejemplo, correr los nodos que hacen procesamiento de datos, o correrlos con una interfaz gráfica programada en otro nodo.

2.2.3. Rviz y Rqt_plot

Rviz y Rqt_plot son herramientas de ROS para visualización de datos. Rviz[8] permite la visualización de datos de sensores, modelos de movimiento de robots, y otros datos tridimensionales en una vista espacial. Rqt_plot[9] es una herramienta para la visualización de datos escalares, que permite de forma rápida tener mediciones de valores en función del tiempo. Ambas herramientas funcionan suscribiéndose a tópicos, de forma que funcionan transparentemente en un sistema, pudiéndose iniciar como nodos en un lanzador al igual que cualquier otro nodo.

2.2.4. Rosbag

Rosbag es una herramienta de ROS que permite guardar mensajes en archivos, permitiendo la reproducción del estado de un sistema posteriormente. Esto permite en un sistema separar el proceso de adquisición de datos del de procesamiento, grabando los datos con Rosbag para adquirir los datos y luego reproduciéndolos. De esta forma un nodo de procesamiento no ve diferencia entre los datos adquiridos directamente (con un sensor, en este caso el radar) de los datos reproducidos con Rosbag.

2.3. Equipos utilizados

2.3.1. Radar

Este es el equipo más importante usado en este trabajo, y consiste en un radar Acumine de 94 GHz. Éste es un radar FMCW y consiste en un dispositivo sensor (emisor y receptor de radio y espejo director) y un computador que procesa los datos obtenidos por el sensor. El sensor se muestra en la Figura 2.6.

Figura 2.6: Radar Acumine.

Este sensor consta de dos subsistemas, uno para el sistema emisor-receptor y otro para el motor del espejo, y estos se controlan independientemente. En términos lógicos, el radar, en particular el computador del radar, funciona con el sistema operativo QNX, y ofrece tres servidores TCP, que se muestran a continuación.

`RadarServer`: Este servidor recibe comandos que controlan el procesamiento de los datos del radar, como los modos de procesamiento de la señal recibida o los parámetros de

la transformada de Fourier que se le aplica a la señal, y entrega los datos de la señal recibida por el radar.

`ScannerServer:` Este servidor recibe comandos que controlan el sensor del radar, como los parámetros del controlador PID[10] del espejo del radar o la pose del espejo rotatorio[6], y entrega datos acerca de la pose del radar.

`ScanningRadarServer:` Este servidor solo entrega datos del radar, combinando los datos de mediciones del radar con los de la pose de éste, realizando extrapolaciones para, dado el *timestamp* de los datos que está recibiendo el radar, calcular la pose del *scanner* y crear un mensaje conjunto.

La interfaz lógica del radar se muestra en la Figura 2.7.

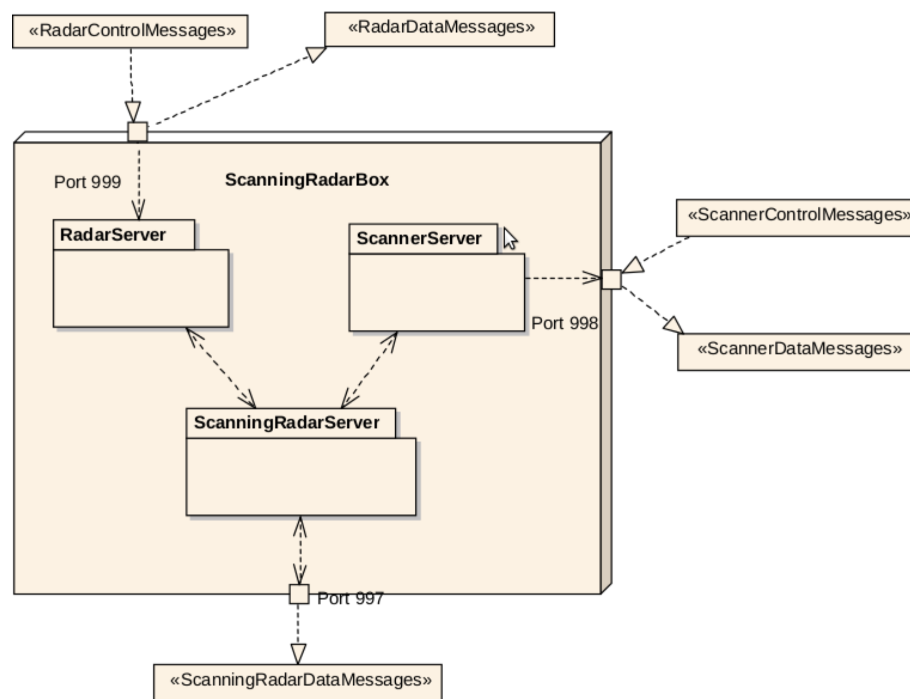


Figura 2.7: Interfaz lógica ofrecida por el radar [1].

En [1] está especificada la estructura de los mensajes recibidos y enviados por el radar, con lo que es posible implementar clientes que se comuniquen con el radar usando esos mensajes y con los que se puedan hacer otros procesamientos de los datos.

Los distintos servidores del radar envían paquetes con la estructura general que se muestra en la Figura 2.8.

[6]Pose se refiere al ángulo relativo del espejo con respecto a alguna recta de visión del radar.

Cabecera	Timestamp en segundos	Timestamp en nanosegundos	Largo de mensaje	Datos	*Checksum*	Cola
0xB255	[8 bytes]	[4 bytes]	[4 bytes]	[variable]	[2 bytes]	0x759F

Figura 2.8: Estructura general de los mensajes enviados por el radar.

La estructura de la Figura 2.8 tiene algunas diferencias con [1], como por ejemplo las definiciones de la cabecera y de la cola. Esta estructura fue descubierta analizando los paquetes que envía el radar[7].

Las especificaciones técnicas del radar relevantes para este trabajo se muestran en la Tabla 2.1.

Frec. portadora	94 GHz
Potencia transmitida	10 mW
Rango de operación	Entre 1 m y 200 m
Frec. de giro	Entre 0 y 5 Hz
Conexión	Ethernet

Tabla 2.1: Especificaciones técnicas del radar utilizado.

2.3.2. Láser

Para comparar el radar con otros sensores se usó un sensor láser LD-LRS1000, de SICK, que se muestra en la Figura 2.9. Este es un sensor láser giratorio, que detecta presencia o ausencia de obstáculos. Este sensor tiene interfaz a ROS, de forma que sus mediciones se pueden ver directamente con Rviz. Sus especificaciones se muestran en la Tabla 2.2.

Frec. de emisión	$3,3126 \cdot 10^{14}$ Hz (905 nm)
Res. angular	0.0625°
Campo de visión	360°
Frec. de giro	Entre 5 Hz y 10 Hz
Rango de operación	Entre 0,5 m y 250 m
Conexión	Ethernet

Tabla 2.2: Especificaciones técnicas del sensor láser utilizado.

Figura 2.9: Sensor láser SICK.

[7]Esto se realizó previo a esta memoria, no por el autor.

2.3.3. Base móvil

También se usó una base móvil para mover los sensores a los puntos de experimentación, la que se muestra en la Figura 2.10. Esta es una base móvil Husky, de Clearpath Robotics. Éste es un vehículo orientado al movimiento en terrenos difíciles (tierra, pasto, lodo) y para el montaje rápido de cargas útiles (en este caso, los sensores), además de proveer energía para éstos. Este dispositivo tiene una interfaz directa a ROS, permitiendo su control desde un computador y pudiendo entregar mediciones como el estado de las baterías, odometría, etc. Sus especificaciones técnicas se muestran en la Tabla 2.3.

Masa	50 kg
Carga máxima	75 kg
Rapidez máxima	$1\ \mathrm{m\,s^{-1}}$
Autonomía	Entre 3 h y 8 h
Conexión	RS-232[1]

Tabla 2.3: Especificaciones técnicas de la base móvil utilizada.

Figura 2.10: Base móvil Husky.

2.4. Experiencias similares y el aporte de esta memoria

En [11] se muestra la implementación de una librería de control para un radar Acumine. Ese radar, a pesar de no ser idéntico al que se usa en esta memoria, funciona de la misma forma que el radar de ésta, con comunicación por protocolo TCP. El software producido en ese trabajo de título es un cliente con interfaz gráfica construida sobre la librería XView que permite controlar los parámetros del radar, como la resolución radial y el controlador del espejo, entre otros, además de mostrar los datos obtenidos por el radar y/o guardar los datos en un archivo.

El aporte de esta memoria, comparada con [11], será el habilitar el radar para su uso con la plataforma ROS, permitiendo de esta forma la compatibilidad del radar con otros dispositivos que se dispongan en el laboratorio (compatibles con ROS), como puede ser otros sensores o actuadores, y permitiendo la reutilización de código de la comunidad de ROS.

[1]No obstante, la base móvil se conectó al computador principal usando un adaptador a USB.

Capítulo 3

Diseño e implementación

En este capítulo se muestran los requerimientos específicos del software implementado, las decisiones de diseño que se tomaron para cumplir esos requerimientos y las características del software implementado.

3.1. Requerimientos

Después de analizar los objetivos de la sección 1.2, se tiene que el software a implementar debe poder hacer lo siguiente:

1. Controlar los parámetros del radar, a decir:
    a) El modo de procesamiento del radar.
    b) Los parámetros de la transformada de Fourier que se aplica a los datos.
2. Controlar los parámetros del *scanner*, a decir:
    a) Los parámetros del controlador PID.
    b) La posición y velocidad del *scanner*.
3. Guardar y cargar configuraciones en el radar.
4. Procesar los datos generados por el radar con un procesador CFAR.
5. Permitir la visualización de los datos que genera el radar.
6. Permitir la comunicación del software implementado con otros nodos de ROS.

3.2. Decisiones de diseño

3.2.1. Lenguajes de programación

La plataforma ROS tiene dos lenguajes principales, Python y C++. Python tiene la ventaja de su expresividad, esto es que se pueden hacer tareas complejas con pocas líneas de código, mientras que C++ es un lenguaje altamente eficiente, de forma que se puede hacer procesamiento de datos rápidamente. Se escogió trabajar con el lenguaje Python para las comunicaciones del radar, porque es más fácil de programar, y porque en esta parte no se requiere alto desempeño, al ser esta parte del software limitada por la entrada de la red. También, por el mismo motivo de simplicidad y desempeño, se escogió Python para implementar la interfaz gráfica del sistema. No obstante, se decidió implementar la funcionalidad de generación de nubes de puntos y detección de obstáculos en C++, ya que esta sección debe ser de alto desempeño, y porque la librería usada para generar los datos a visualizar, que se verá posteriormente, está hecha para C++.

3.2.2. Interfaz a ROS

Como se vio en la sección 2.2, ROS ofrece dos formas de comunicación entre nodos, con mensajes y con servicios. Por otro lado, el radar tiene dos tipos de comunicación.

- Los datos que el radar genera, que van desde el radar hacia el cliente.
- Las acciones de configuración del radar, que van desde el cliente hacia el radar.

Analizando las herramientas de comunicaciones en ROS, las decisiones tomadas para la interfaz del sistema fueron las siguientes:

Servicios para los mensajes de control del radar, porque, al ser los servicios mensajes pequeños en comparación con los que envía el radar y ser usados con baja frecuencia, no se necesita que los nodos estén permanentemente consultando si se ha enviado una señal de control, como pasaría si se usaran mensajes y tópicos. Sin embargo, el radar no envía ninguna clase de realimentación acerca de cómo se han recibido los datos, de forma que solamente se usan los servicios como solicitudes, sin utilizar las respuestas que incorporan los servicios. Las definiciones de servicios son directas del protocolo [1] y se mapean con los mensajes de control del radar. De esta forma, la serialización que ofrece ROS por defecto permite que un mensaje serializado pueda ser enviado por la red sin mayor procesamiento dentro del paquete. No obstante, un paquete serializado por ROS no está listo para ser enviado por la red. Se debe también agregar las cabeceras y el *checksum* antes de poder enviarlo por la conexión TCP.

Tópicos para los datos enviados por el radar, ya que, siendo estos mensajes los más pesados del sistema, es necesario que los nodos estén constantemente obteniendo mensajes del radar

y publicándolos como mensajes de ROS. Las definiciones de mensajes (en la carpeta `msg`) no reflejan exactamente la estructura de los paquetes, ya que los paquetes enviados por el radar tienen datos inútiles para el sistema, es decir el usuario del paquete, como el *checksum* y el largo de las listas de submensajes.

3.2.3. Interfaz gráfica

El sistema requiere una interfaz gráfica. Dado que son relativamente pocas las acciones a hacer con esta interfaz, se prefirió la simplicidad de programación por sobre la cantidad de características ofrecidas por la librería. Se analizaron dos librerías:

Tkinter [12]: Esta es la librería estándar (de facto) de interfaces gráficas para Python, basada en el *toolbox* Tk del lenguaje Tcl. Viene por defecto con Python.

WxPython [13]: Esta es una librería de interfaces gráficas basada en WxWidgets, una librería de interfaces gráficas para C++. Se debe instalar aparte del intérprete de Python.

Considerando que la interfaz no es crítica para el sistema, para escoger se programó un bosquejo de la interfaz gráfica en ambos lenguajes y se vio cuál era más cómoda de programar. Con ello se escogió WxPython.

3.2.4. Visualización de datos

El sistema tiene que poder hacer visualización de los datos que genera el radar. Para esto se tienen dos opciones:

- Tomar los datos que emite el radar directamente y hacer un visualizador para estos.
- Adaptar los datos que genera el radar a un formato visualizable por Rviz.

En este caso es más simple adaptar los datos para Rviz. Para esto, se usó el formato de nubes de puntos, que es el mismo formato que usa el sensor Kinect [14], usando la librería *Point Cloud Library* (PCL) [15].

3.3. Estructura general del software

La librería implementada en esta memoria es, en su conjunto, un cliente que se conecta al radar para obtener mediciones de éste y enviarle señales de control. Por esto la librería se le llama “cliente” en otras partes de esta memoria. La librería se implementó siguiendo una estructura de nodos que refleje a la de los servidores del radar. Es decir, se implementó un

nodo para el `RadarServer`, uno para el `ScannerServer` y uno para el `ScanningRadarServer`, llamados `radar_server`, `scanner_server` y `scanning_radar_server`, respectivamente. El trabajo de cada uno de estos nodos es responder a las llamadas de los servicios de control del radar, generando los paquetes de control especificados en el protocolo [1], a la vez que leer continuamente los datos que envía el radar, procesarlos y publicar mensajes en los tópicos correspondientes. Esta estructura se muestra en la Figura 3.1.

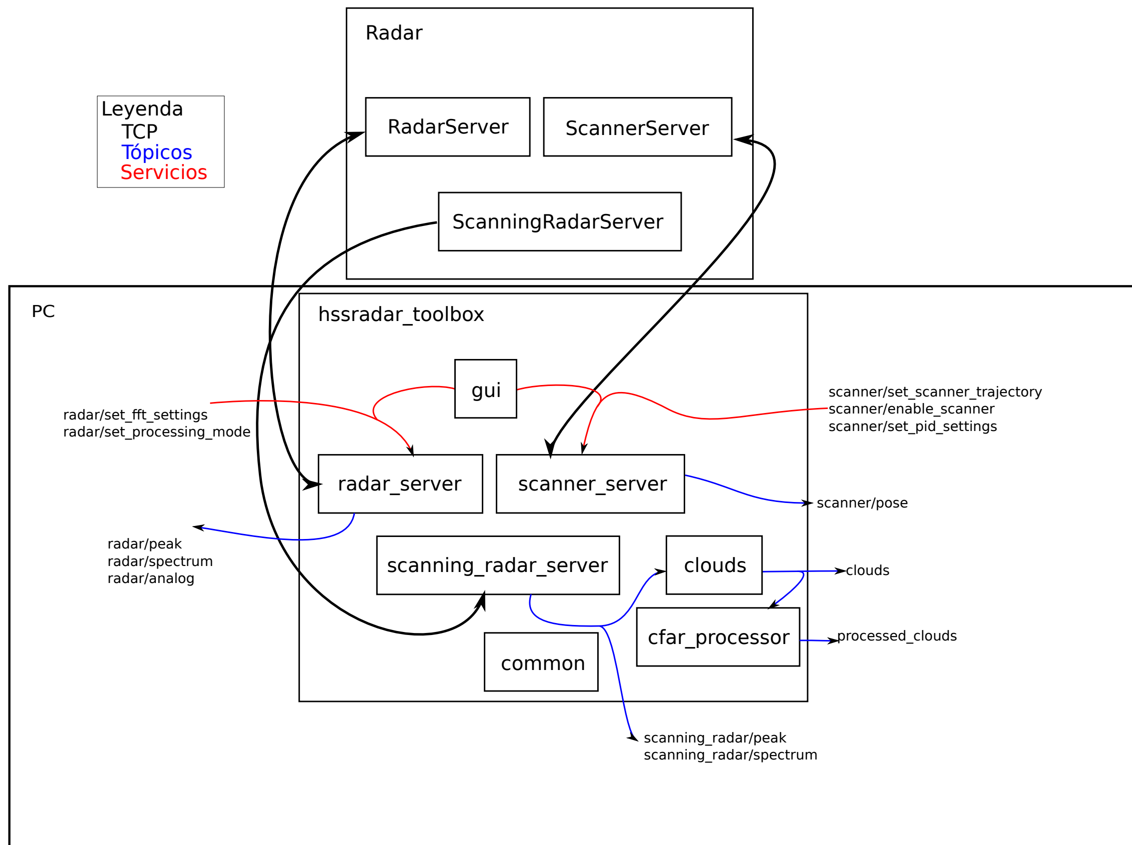


Figura 3.1: Estructura general de los nodos implementados y su interfaz a ROS.

Esta estructura muestra las tres áreas de funcionamiento de los nodos:

- Nodos de comunicaciones: Estos son `radar_server`, `scanner_server` y `scanninng_radar_server`, los nodos que establecen y mantienen la conexión TCP entre el radar y el computador cliente, y que se encargan de ofrecer los servicios de configuración y los tópicos de datos.
- Nodos de procesamiento y visualización: Estos son `clouds` y `cfar_processor`.
- `gui`. Este nodo es la interfaz gráfica del sistema.

También se implementó el módulo `common`, que no es un nodo de ROS, sino que es una librería con algunas funciones necesarias por los tres nodos de comunicaciones, como la envoltura de paquetes con cabecera y *checksum* y los formatos de los paquetes.

3.4. Decisiones incidentales

En la sección 3.2 se muestran las decisiones tomadas antes de la implementación de la librería. En esta sección se muestran los problemas que aparecieron en la implementación y cómo se abordaron.

3.4.1. Lectura y escritura de mensajes

En Python, una forma usual de convertir flujos de datos en bytes a datos con significado para el programa es usar el módulo `struct`, que permite conversiones directas entre datos serializados y datos usables por un programa. En este caso, los paquetes son de largo variable, ya que las configuraciones del radar definen la cantidad de datos que se envían por mensaje.

Dado esto, se requirió implementar una forma de parsear paquetes que permitiera paquetes de largo variable. Se implementó la clase `NetParser`, en el módulo `common`, que permite la conversión de listas de datos de largo variable, en tanto se tenga previamente el largo del paquete a convertir. Esta clase permite la descripción de una cadena de bytes como una cabecera, una lista de largo variable y una cola, cada una de ellas con un formato característico. El largo de la lista está determinado por uno de los datos obtenidos de la cabecera, permitiendo así que, dada una cadena, se puedan obtener un número indeterminado de datos de ésta.

Para las señales de control, que van de los nodos al radar, se hizo una función en el módulo `common` que envuelve un mensaje (que en este caso sería la solicitud de un servicio) de ROS en las cabeceras y colas correspondientes, y se envía el paquete. Definiendo los campos de los servicios en el mismo orden que el protocolo, basta solamente la serialización ofrecida por ROS para que los mensajes sean serializados en el orden y con los tamaños correctos.

3.4.2. *Sockets* y sincronización

Un motivo para decidir que los nodos estuvieran organizados de la misma forma que los servidores del radar fue el hecho de que de esta forma, cada nodo tendría un *socket*, y así no habría que tener más de un nodo usando un *socket*. Entonces, para cada nodo se definió una variable global (a nivel de módulo) para el *socket*, de forma que cada nodo crea la conexión al iniciarse, mientras que los *handlers* de los servicios y el hilo de lectura de mensajes usan el *socket*. Ahora, como en cada nodo se tiene un hilo leyendo datos del radar continuamente, se necesita una forma de sincronización que permita a los servicios usar la conexión para enviar sus datos. Para ello se agregó como variable global un `Lock` asociado al *socket* del nodo. De esta forma, el hilo de lectura de mensajes, cuando va a leer un mensaje, toma el `lock`, usa el *socket*, y después de que lee los datos, lo libera, mientras los *handlers* de los servicios hacen lo mismo.

3.4.3. Robustez de la conexión

En las primeras implementaciones del sistema, éste fallaba por problemas de conexión con el radar. Se vio que, en ciertos momentos, al hacer cambiar de modo de procesamiento de datos al radar, éste reiniciaba la conexión. Como las primeras implementaciones no eran capaces de detectar este problema, entonces, al reiniciarse la conexión, el nodo `radar_server` se caía completamente, en momentos aleatorios. Esto se compensó agregando un chequeo en las lecturas de datos, que reinicie la conexión en caso de que no se tengan datos en la conexión actual. Un problema que esto puede causar es que se podría perder sincronía en los paquetes, es decir, que al reconectarse, se comiencen a recibir paquetes que no comiencen con las cabeceras correctas. Sin embargo, en las pruebas realizadas, se vio que el radar siempre envía paquetes con las cabeceras correctas, sea cual sea el momento de la conexión, entonces el reinicio de la conexión no genera problemas de sincronización de paquetes.

3.4.4. Parámetros del radar y configuración

Como el radar no da información de sus parámetros directamente, es necesario que, entre un uso y otro del radar, se guarde el estado[1] de éste. Para ello, se usó el servidor de parámetros de forma que, cada vez que un servicio es llamado, se pone en el servidor de parámetros la configuración deseada. Estos parámetros son guardados en un archivo de configuración, el cual es cargado en el servidor de parámetros cuando inicia el cliente, antes de que inicien los nodos. Cuando los nodos inician, éstos toman los parámetros que están definidos y llaman automáticamente sus servicios con esos parámetros, de forma que al iniciar el cliente, éste configura el radar inmediatamente, poniendo el radar en el mismo estado que estaba cuando se usó por última vez.

3.4.5. Medición de datos desde el radar

El protocolo de comunicación del radar no especifica ninguna clase de realimentación para las señales de control, de forma que, aunque el cliente emita el paquete correcto, no es posible saber directamente si el paquete fue procesado en el radar o siquiera recibido. La única realimentación que provee el radar está en los mismos datos que este envía, que son:

- El modo de procesamiento del radar, que se puede obtener del campo `message_id` de los mensajes.
- El largo de la transformada de Fourier utilizada, que se puede obtener del largo de los mensajes en modo de espectro de frecuencia.
- El ángulo del *scanner*, que se puede obtener de los mensajes del `scanner_server`.

[1]Por "estado" se hace referencia a la rapidez y posición del *scanner*, los parámetros del controlador PID, y el modo de procesamiento y los parámetros de la transformada de Fourier del radar.

- La rapidez del *scanner*, que se puede obtener de los mensajes de `scanner_server`.

También hay datos que se pueden ver a partir de las trayectorias del radar, como los cambios en el control PID del *scanner*, pero que no se envían explícitamente por la conexión. Para poder obtener esos datos del radar, se hizo una suscripción temporal a los tópicos del radar y el *scanner* para poder obtener toda la información posible de los datos enviados en ese momento y ponerla en el servidor de parámetros. Es importante remarcar que el radar no envía toda la información automáticamente: Por ejemplo, cuando el radar está en modo de procesamiento de *peaks*, éste no entrega información sobre el largo de la transformada de Fourier, con lo que no es posible obtener todo el estado del radar en todo momento.

3.5. Funcionalidades implementadas

3.5.1. Nodos de comunicaciones

Tres nodos fueron implementados para la funcionalidad de comunicaciones: `radar_server`, `scanner_server` y `scanning_radar_server`, que cumplen las siguientes funciones:

- Responder a las llamadas a los servicios, que son básicamente tomar el mensaje del servicio, envolverlo en con la cabecera y cola correspondiente, y enviarlo a través de la conexión establecida.

- Leer los datos enviados por el radar y publicarlos en los tópicos correspondientes. Para ello el procedimiento es el siguiente:

    1. Se lee una cabecera, de tamaño fijo.

    2. De la cabecera, se obtiene el largo del paquete.

    3. Se obtienen los datos del paquetes.

    4. Se obtiene la cola del paquete, de largo fijo. A partir de este momento el algoritmo trabaja con una cadena fija, y no es necesario leer más de la conexión.

    5. Se verifica el *checksum* del paquete. Si esta correcto, se continúa, sino, se descarta el paquete y se vuelve al inicio.

    6. Se decide el tipo del mensaje. Con esto, se despacha al `NetParser` correspondiente.

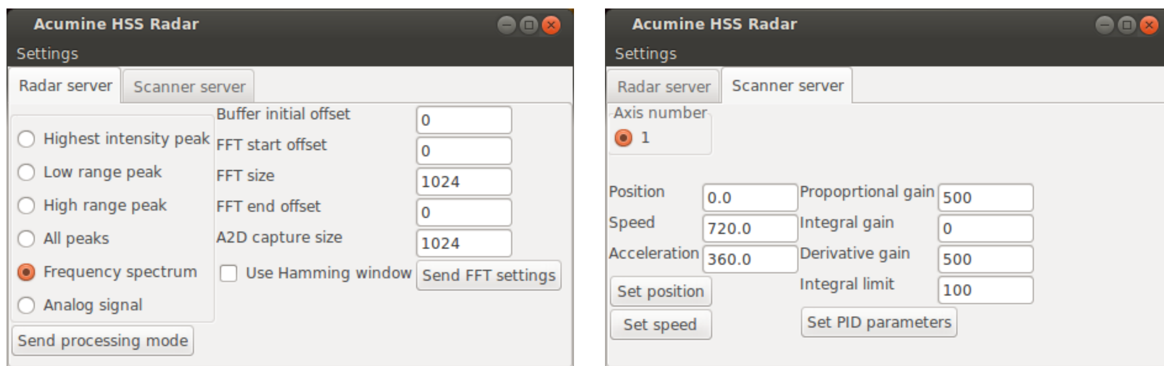
    7. Dado el tipo del mensaje, se publica en el tópico correspondiente.

ROS ofrece paralelismo en los servicios, de forma que solo es necesario implementar las funciones, y la plataforma se encarga de lanzar un hilo cuando un servicio es llamado. Sin embargo, la función que lee datos de la conexión se lanza en un hilo separado, ya que este

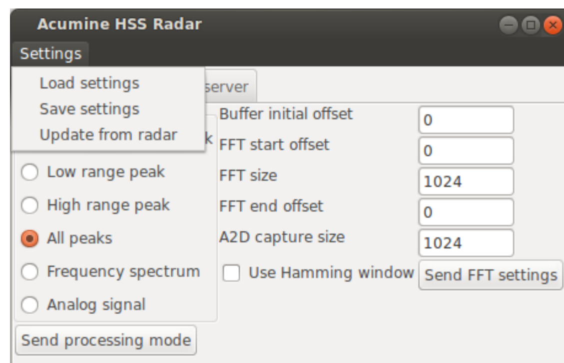
hilo está funcionando constantemente, al contrario de los *handlers* de servicios.

3.5.2. Interfaz gráfica

A pesar de que el radar es completamente controlable por línea de comandos con el programa `rosservice`, que permite llamar servicios directamente, se hace necesario una interfaz que permita configurar el radar de una forma más simple. Se creó una interfaz gráfica para el control del radar que permite configurar todos los detalles de éste. Esta interfaz se muestra en la Figura 3.2.



(a) Controles del radar (b) Controles del *scanner*



(c) Menu principal

Figura 3.2: Interfaz gráfica del sistema implementado.

Esta interfaz permite configurar todos los aspectos que ofrece el radar, esto es:

- Datos del radar, en la pestaña `Radar server`:
  - El modo de procesamiento del radar.
  - Los parámetros de la transformada de Fourier.
- Datos del *scanner*, para cada grado de libertad (eje) del espejo, en la pestaña `Scanner server`:

- La posición y rapidez de giro del espejo en el eje.
- Los parámetros del controlador PID del eje.

Nótese que en el caso del `scanner_server`, a pesar de que el sistema permite elegir un eje, éste solo tiene uno, con lo que solo se tiene una opción para el eje. No obstante, el protocolo [1] especifica mensajes con más de un grado de libertad, y la implementación respeta esa posibilidad, permitiendo en el futuro agregar más grados de libertad al *scanner* (por ejemplo, montando el radar en una plataforma giratoria) sin modificaciones mayores al programa.

El menú `Settings` ofrece tres funcionalidades:

- Cargar un archivo de configuración. Esto carga un archivo `yaml` en el servidor de parámetros y luego llama a los servicios de los nodos de comunicaciones con los nuevos parámetros para fijarlos en el radar.
- Guardar configuración. Esto toma todos los parámetros del servidor de parámetros y los pone en un archivo `yaml`.
- Obtener datos del radar. Esto toma todos los datos que puede del estado del radar y los pone en la interfaz gráfica.

Cuando la interfaz gráfica se cierra, ésta guarda automáticamente todos los parámetros del radar en un archivo ubicado en `config/acumine_radar.yaml`, independientemente de que se haya usado la funcionalidad de guardar configuración. De esta forma es posible obtener éstos al inicio del programa. Es importante notar que la interfaz gráfica se comunica con el radar mediante llamadas a servicios de los nodos de comunicaciones, de la misma forma que se puede hacer mediante línea de comandos con el programa `rosservice`. Es decir que las comunicaciones de la interfaz gráfica son hechas puramente con ROS. Finalmente, por lo mismo, se tiene que este nodo es prescindible para el funcionamiento del sistema, ya que todas las llamadas a servicios se pueden hacer con `rosservice`.

3.5.3. Nubes de puntos

Usando la estructura de mensajes definida por los nodos de comunicaciones, se implementó el nodo `clouds`, que recibe mensajes del nodo `scanning_radar_server`, acumulándolos hasta que el *scanner* completa una vuelta, y convirtiéndolos a una nube de puntos que representa la vista completa del radar. De esta forma, se pueden trabajar los datos de forma estándar, pudiéndose usar las funcionalidades de PCL con los datos del radar. Además, en este formato se pueden ver fácilmente los datos con la herramienta Rviz. Nótese que el nodo `clouds` no lee directamente de la conexión TCP con el radar, sino que lee de los nodos de comunicaciones. Esto permite separar el proceso de adquisición de datos crudos y el procesamiento de estos como nubes de puntos, de forma que es posible tomar datos con el radar y luego, con Rosbag, reproducirlos y convertirlos a nubes de puntos para su procesamiento.

También, en caso de que el radar no esté en movimiento, este nodo publica nubes de puntos correspondientes al ángulo al que el *scanner* esté mirando.

3.5.4. Detección de obstáculos

Para la funcionalidad de detección de obstáculos, se implementó el nodo `cfar_processor`, que aplica un procesador CFAR a una nube de puntos. Para esto se usó la librería rápida de vecinos más cercanos aproximados (*Fast Library for Approximate Nearest Neighbors*, FLANN), que forma parte de PCL. Esta librería ofrece una estructura de datos que permite hacer una búsqueda de vecinos más cercanos usando *kd-trees*, lo cual es básicamente una forma de encontrar puntos vecinos usando divisiones del espacio donde estos se encuentran. Con esto, esta implementación permite que el test del procesador CFAR tome puntos en un anillo tridimensional, al contrario de las implementaciones estándar de CA-CFAR, que solo toman un *A-scope* (unidmensional), permitiendo así que el test que se hace para cada punto considere más información, con lo que se obtiene una mejor estimación de los parámetros de la distribución estadística del ambiente. La salida de este nodo son nubes de puntos de obstáculos, de la misma forma que se puede hacer cuando el radar está en modo de *peaks*. Sin embargo, el valor de intensidad de los puntos de las nubes que publica este nodo no es la intensidad del espectro para el punto, sino que la probabilidad de detección de cada punto. También, para efectos de comparación, se implementó una versión de CFAR unidimensional.

3.5.5. Lanzadores

Considerando que todos los nodos implementados son programas ejecutables por si mismos, se vuelve complicado iniciar el sistema completo, con las funcionalidades necesarias para lo que se quiere hacer. Para facilitar el uso de los nodos implementados, se crearon los siguientes lanzadores:

`radar.launch`: Este lanzador se encarga de lanzar los nodos de comunicaciones, junto con la interfaz gráfica. La interfaz gráfica puede ser omitida con la opción `with_gui:=false` como argumento del lanzador. Este lanzador es básico, ya que también carga los parámetros necesarios para que los nodos funcionen, como la IP del computador del radar y el rango del sensor, los cuales son necesarios para los nodos de procesamiento y visualización.

`clouds.launch`: Este lanzador se encarga de lanzar los nodos de procesamiento de los datos. Además lanza una instancia de Rviz que permite visualizar las nubes de puntos cuando se reciben.

3.6. Modos de uso del sistema

Los lanzadores descritos en la sección anterior fueron pensados para usos como éstos:

- **Configuración de parámetros**

  1. Se llama al lanzador `radar.launch`.

     ```
     >roslaunch hssradar_toolbox radar.launch
     ```

  2. Se configura el modo de procesamiento, los parámetros de la transformada de Fourier, la posición o rapidez del *scanner*, etc.

  3. Se cierra la interfaz gráfica, cerrando así todo el sistema y guardando la configuración en la ubicación por defecto, o se usa la funcionalidad de guardar configuración para guardar los parámetros del radar en otro archivo.

- **Captura de datos**

  1. Se llama al lanzador `radar.launch`.

     ```
     >roslaunch hssradar_toolbox radar.launch
     ```

  2. Se configura el sistema, con un archivo de configuración o usando la interfaz gráfica.

  3. Se graban datos con Rosbag.

     ```
     >rosbag record /acumine_radar
     ```

- **Captura y visualización de datos**

  1. Se llama al lanzador `radar.launch`.

     ```
     >roslaunch hssradar_toolbox radar.launch
     ```

  2. Se configura el radar y se deja el *scanner* girando a rapidez fija.

  3. Se llama al lanzador `clouds.launch`.

     ```
     >roslaunch hssradar_toolbox clouds.launch
     ```

  4. Se visualizan los datos con Rviz.

  5. Se graban datos con Rosbag.

     ```
     >rosbag record /acumine_radar
     ```

Con estos modos se hace uso de todas las características del software implementado, cubriendo todos los requerimientos del software.

Capítulo 4

Resultados

En este capítulo se describen las pruebas que se hicieron con el radar, con las que se pretende mostrar todas las funcionalidades del radar a las que se tiene acceso gracias al cliente implementado. Estas pruebas fueron hechas con datos reales, es decir, datos generados con mediciones del radar.

4.1. Desempeño en la red

Para observar el uso de la red por parte del radar, se usó el programa `Wireshark` para tomar una medida del ancho de banda ocupado por sus servidores. Se tomó una prueba de 1 minuto, capturando los paquetes de los distintos puertos. Los resultados se muestran en la Tabla 4.1.

Servidor	**Mensaje**	**Ancho de banda** ($\mathrm{MB\,s^{-1}}$)
`RadarServer`	*Peak* más potente	0,635
	Peak más cercano	0,635
	Peak más lejano	0,635
	Todos los *peaks*	0,473
	Espectro en frecuencia	7,887
`ScannerServer`	Pose	0,128
`ScanningRadarServer`	Modo de *peaks*[1]	1,052
	Espectro en frecuencia	8,378

Tabla 4.1: Ancho de banda ocupado por cada servidor en distintos modos de operación del radar.

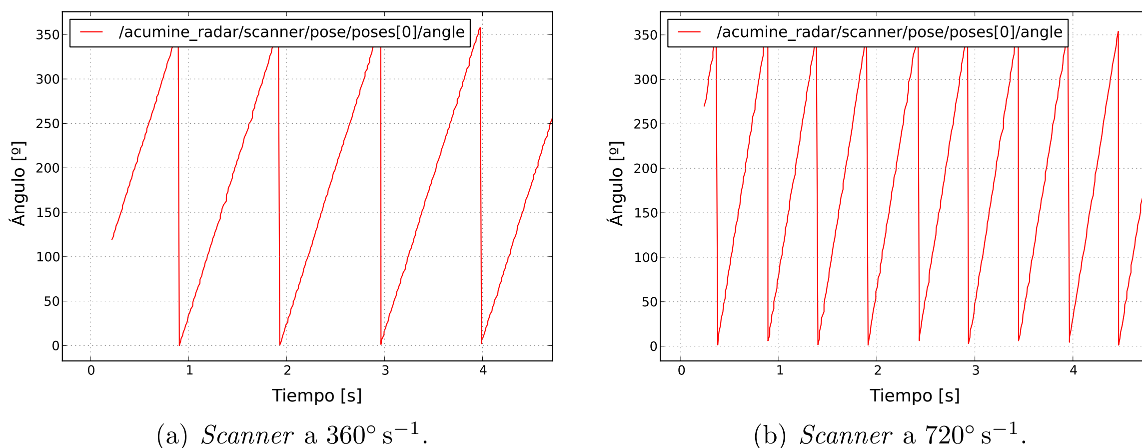
[1]Considerando que este servidor solo extrapola y concatena mensajes de los otros dos, este modo se probó solamente con el modo de todos los *peaks* del `RadarServer`, ya que es el modo cuyos mensajes son más largos.

En la Tabla 4.1 se ve que en los distintos modos de *peaks* el ancho de banda ocupado por el `RadarServer` es menor que el ocupado en modo de espectro en frecuencia. Esto se da porque los mensajes de espectro en frecuencia son mucho más grandes que los otros. En el `ScannerServer` se ve que los mensajes de Pose ocupan el menor ancho de banda de todos, porque son los más pequeños que envía el radar. En el `ScanningRadarServer` se ve que como los mensajes de este servidor son compuestos, este servidor es el que ocupa más ancho de banda de los tres, siendo el modo más pesado el de espectro en frecuencia.

Un potencial problema con respecto a el desempeño del cliente implementado es que el hecho de que éste pudiese no tener suficiente *troughput* para la cantidad de datos que recibe, es decir, que la librería implementada no fuese capaz de procesar los datos a una velocidad igual o superior a la que el radar los envía. Sin embargo, usando la herramienta de Linux `/proc/net/tcp`, que entrega información sobre las conexiones TCP del sistema, se vio que el largo del *buffer* de recepción en los puertos a los que el cliente se conecta con el radar es cero, con lo que se ve que el cliente implementado tiene un *throughput* aceptable para el radar.

4.2. Pruebas de *scanner*

Las pruebas hechas para probar el *scanner* consistieron en pruebas de control de posición y rapidez. Primero se probó la funcionalidad de giro a rapidez constante, después de configurar la rapidez del *scanner* en $360° \mathrm{s}^{-1}$ y $720° \mathrm{s}^{-1}$ respectivamente. Los datos de posición del *scanner* se visualizaron con rqt_plot[1], como se ve en la Figura 4.1.



(a) *Scanner* a $360° \mathrm{s}^{-1}$. (b) *Scanner* a $720° \mathrm{s}^{-1}$.

Figura 4.1: Control de rapidez del *scanner*.

Un problema que se ve en la Figura 4.1 es que la rapidez alcanzada por el radar en ambos casos no corresponde a la asignada mediante el control del *scanner*, sino que es más

[1] Rqt_plot grafica los datos en tiempo real, desde que se inicia el programa. Esto significa que en las pruebas realizadas el tiempo no inicia de cero.

lenta. Según los mismos datos que envía el radar, la rapidez alcanzada en cada caso es de 351,5625 $^\circ\,\mathrm{s}^{-1}$ y 703,125 $^\circ\,\mathrm{s}^{-1}$ respectivamente. Esto no se pudo arreglar con los parámetros del controlador.

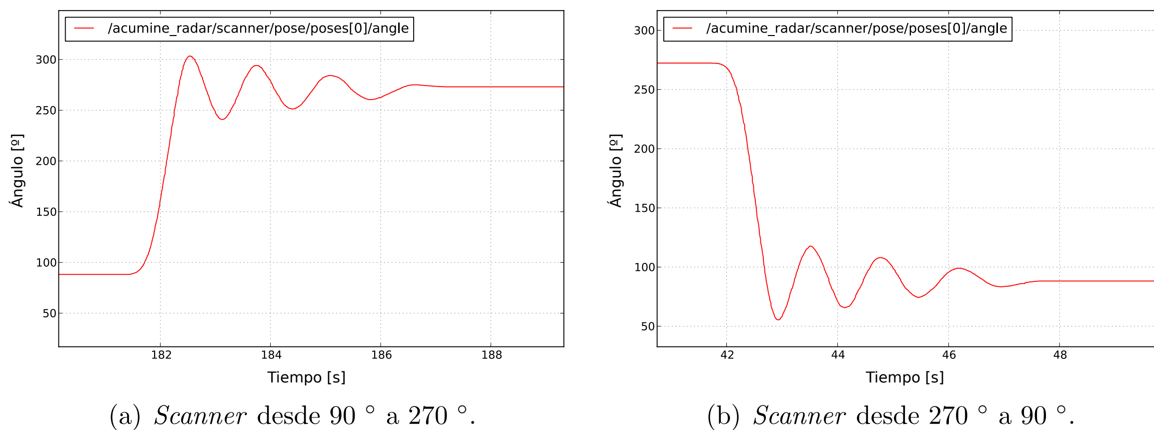
Luego se probaron las funcionalidades de control de posición del *scanner*. Para ello se hizo moverse el *scanner* desde la posición fija de 90 ° a 270 ° (prueba conocida como *respuesta al escalón*) y viceversa con distintas configuraciones del controlador PID. En las pruebas se usó la siguiente configuración de posición y rapidez:

- Rapidez inicial: 360 $^\circ\,\mathrm{s}^{-1}$
- Aceleración inicial: 360 $^\circ\,\mathrm{s}^{-2}$

Primera configuración:

- Ganancia proporcional: 50
- Ganancia integral: 0
- Ganancia derivativa: 0
- Límite de integral: 500

Los resultados se muestran en la Figura 4.2.



(a) *Scanner* desde 90 ° a 270 °.

(b) *Scanner* desde 270 ° a 90 °.

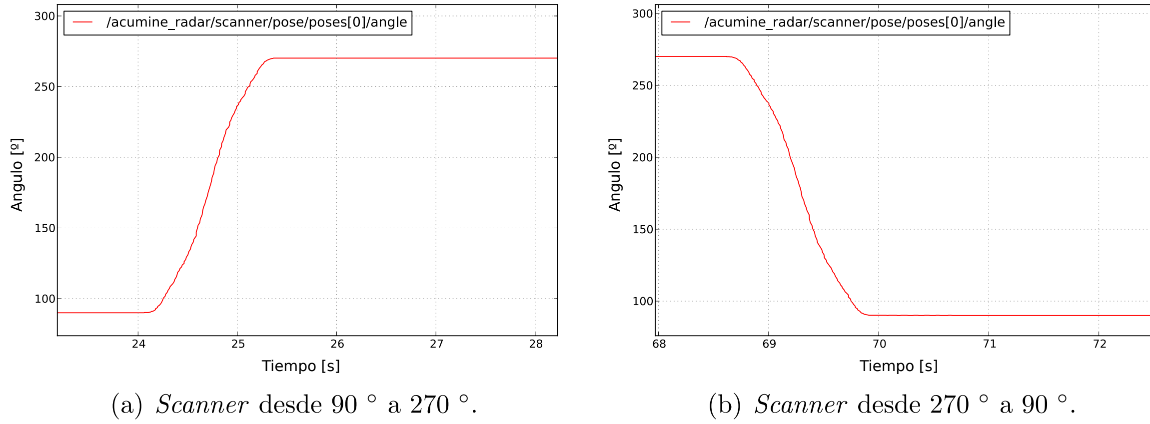
Figura 4.2: Primera prueba de control de posición del *scanner*.

Segunda configuración:

- Ganancia proporcional: 500
- Ganancia integral: 0
- Ganancia derivativa: 0

- Límite de integral: 500

Los resultados se muestran en la Figura 4.3.



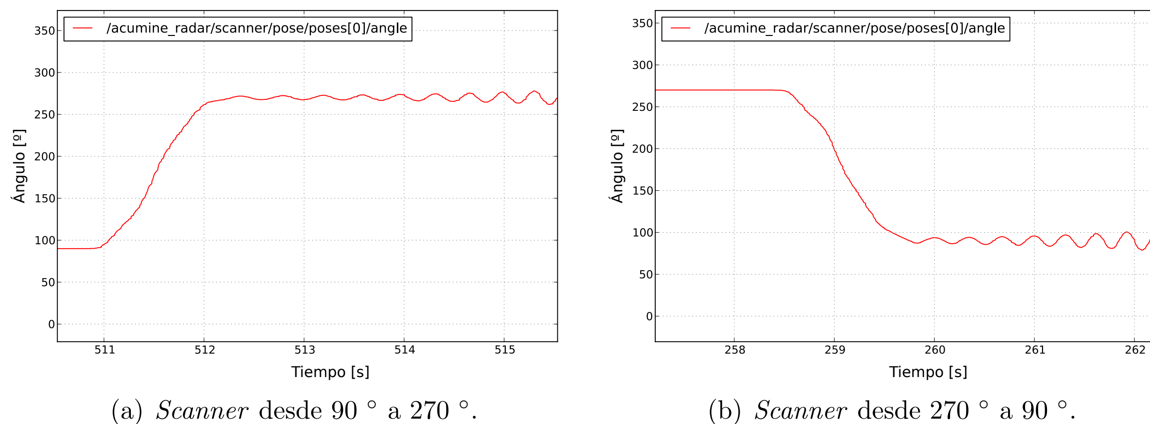
(a) *Scanner* desde 90 ° a 270 °. (b) *Scanner* desde 270 ° a 90 °.

Figura 4.3: Segunda prueba de control de posición del *scanner*.

Tercera configuración:

- Ganancia proporcional: 500
- Ganancia integral: 150
- Ganancia derivativa: 500
- Límite de integral: 500

Los resultados se muestran en la Figura 4.4.



(a) *Scanner* desde 90 ° a 270 °. (b) *Scanner* desde 270 ° a 90 °.

Figura 4.4: Tercera prueba de control de posición del *scanner*.

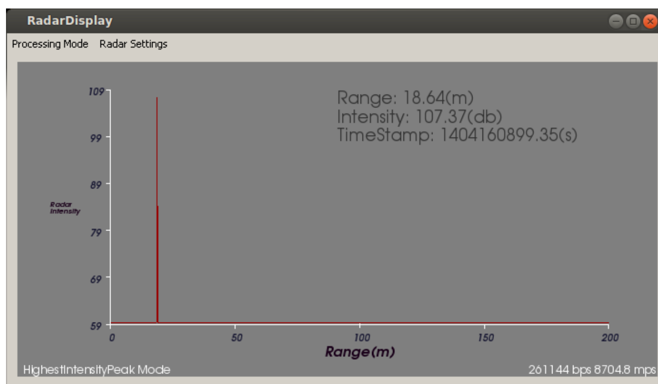
En las Figuras 4.2, 4.3 y 4.4 se muestra la respuesta al escalón del *scanner* con diversas configuraciones del controlador. En la Figura 4.2 se ve que hay sobreoscilación, esto es que

el *scanner* alcanza el punto deseado, pero se excede y comienza a oscilar en torno a éste, con una oscilación de amplitud decreciente, estabilizándose en torno al punto deseado. Esto se da porque el controlador tiene bajos valores, en particular tiene bajos valores de ganancia proporcional. En la Figura 4.3 se ve que al aumentar la ganancia proporcional el sistema baja la sobreoscilación y llega (de forma estable) más rápidamente al punto deseado. En la Figura 4.4 se ve que al agregar ganancia integral al controlador, este oscila en torno al punto deseado con amplitud creciente, volviéndose inestable.

Con esto se muestra que el sistema implementado para el *scanner* puede manejar el controlador de éste, y que el controlador funciona de acuerdo a la teoría[10]. Por otra parte, se logró visualizar el comportamiento del *scanner* con herramientas genéricas de ROS, con lo que se tiene la integración del *scanner* a la plataforma.

4.3. Pruebas de radar

El radar fue probado en el laboratorio, y se contrastaron los datos generados por el software implementado y los datos. Para hacer esto se colocó el radar apuntando a una pared. Luego se registraron los datos crudos del radar (paquetes TCP), y se creó un servidor local para hacer que un computador emule el comportamiento del radar emitiendo un solo paquete repetidas veces. Con esto se comparó lo que muestra el cliente oficial con los datos que genera el software implementado. Los resultados se muestran en las Figuras 4.5, 4.6, 4.7, 4.8 y 4.9.

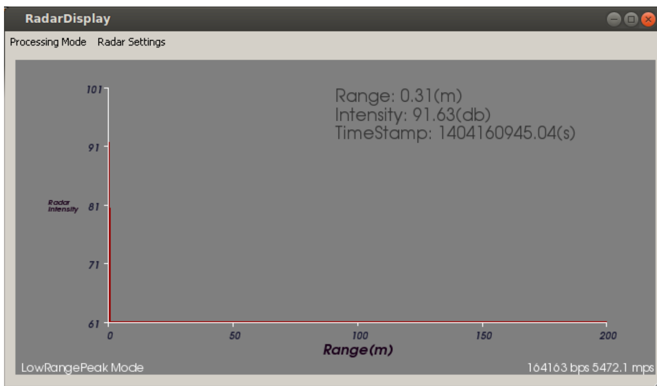


(a) Datos de cliente original.

Pot. media (dB)	59,4019050598
Rango (m)	**Potencia** (dB)
18,6415061951	107,369819641

(b) Datos de mensajes de ROS.

Figura 4.5: Modo de *peak* más potente.



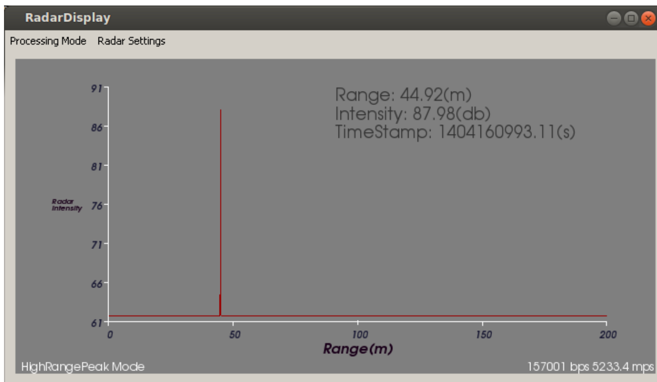
(a) Datos de cliente original.

Pot. media (dB)	61,0440483093

Rango (m)	**Potencia** (dB)
0,310037493706	91,6334381104

(b) Datos de mensajes de ROS.

Figura 4.6: Modo de *peak* más cercano.



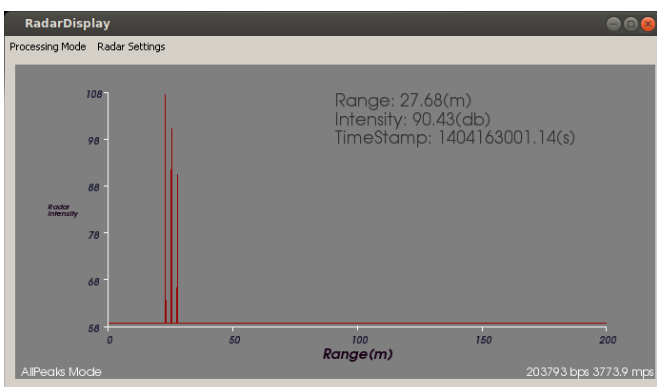
(a) Datos de cliente original.

Pot. media (dB)	61,7037620544

Rango (m)	**Potencia** (dB)
44,9203453064	87,9813461304

(b) Datos de mensajes de ROS.

Figura 4.7: Modo de *peak* más lejano.



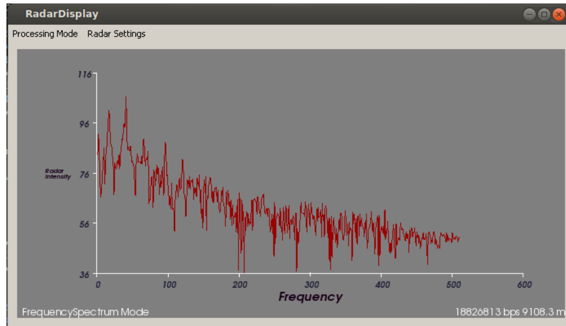
(a) Datos de cliente original.

Pot. media (dB)	58,6520576477

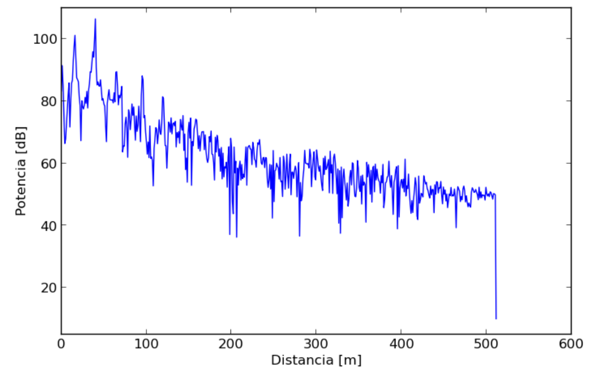
Rango (m)	**Potencia** (dB)
7,23777627945	100,136260986
18,6166133881	107,512908936
30,1995658875	91,367805481
44,8474655151	100,136260986

(b) Datos de mensajes de ROS.

Figura 4.8: Modo de todos los *peaks*.



(a) Datos de cliente original.



(b) Datos de mensajes de ROS.

Figura 4.9: Modo de espectro en frecuencia.

En las Figuras 4.5, 4.6, 4.7, 4.8 y 4.9 se compara el funcionamiento del software oficial del radar y los datos generados por la librería implementada en esta memoria. En las Figuras 4.5, 4.6 y 4.7 se tiene una coincidencia directa entre los datos de los nodos implementados y lo que muestra el software oficial, sin embargo, en la Figura 4.8 no se tiene una coincidencia directa entre ambos. No se logró replicar el comportamiento del software oficial en este caso. Sin embargo, un análisis del protocolo vuelve plausibles los datos obtenidos, ya que están de acuerdo con los datos de los otros modos (como los *peaks* en 18 m y 44 m), y con la configuración misma del experimento, ya que el *peak* a 7 m coincide con la distancia entre el radar y la pared. Finalmente, en la Figura 4.9 se tiene que los datos que muestra el software oficial y los obtenidos por los nodos son iguales, excepto por el último. Dado que el protocolo [1] no se refiere a nada especial en torno a ese valor (que en todas las mediciones realizadas resulto ser 10), este dato se conserva solo por completitud.

4.4. Prueba en terreno

El radar fue probado en la elipse del parque O'Higgins, usando un sensor láser como punto de comparación y una base móvil para poder mover el conjunto completo. La configuración del sistema y el lugar de las pruebas se muestra en las Figuras 4.11 y 4.10 respectivamente.

Figura 4.10: Ubicación del lugar de pruebas. Los puntos en rojo muestran los lugares donde se ubicó el radar para hacer las mediciones. Fuente: Google Maps.

Figura 4.11: Configuración del sistema utilizada en las pruebas.

Las pruebas en terreno fueron hechas en el parque O'Higgins por su cercanía al laboratorio y porque, como el radar tiene un rango de medición fijo de alrededor de 240 m, se debe probar en un lugar suficientemente grande y vacío como para hacer que los obstáculos que el radar pueda detectar estén a una distancia considerable para el rango de éste. El laboratorio es un lugar demasiado pequeño y lleno de obstáculos para eso.

4.4.1. Nubes de puntos

Para la visualización de datos se utilizó la funcionalidad de nubes de puntos, con la que se probaron los distintos modos de procesamiento del radar y se visualizaron en Rviz (a través del nodo `clouds`). Los resultados se muestran en las Figuras 4.12, 4.13, 4.14, 4.15. El modo de espectro en frecuencia, al ser usado en más detalle, se muestra en la Figura 4.16, en la siguiente sección.

En las Figuras 4.12, 4.13, 4.14 y 4.15 se muestran las nubes de puntos generadas con el radar en los modos de *peaks*. Se ve que se da lo mismo que en la prueba de laboratorio, es decir que en el modo de *peak* más cercano las mediciones son de menor rango que en el modo de *peak* más lejano. Un detalle relevante, que puede ser visto como un problema en el diseño del protocolo de comunicación del radar [1] (cuya modificación está fuera del alcance de esta memoria), es que para el `ScanningRadarServer` se especifica que los mensajes de modo de *peaks* solo tienen un *peak* asociado. No obstante, como se puede ver en la Figura 4.15, ciertos ejes tienen más de un *peak*. Esto es porque el `ScanningRadarServer` envía paquetes con *peaks* en distintos rangos, pero con la misma pose, generando redundancia en los datos enviados. En la Figura 4.16 se ve el espectro en frecuencia capturado por el radar, y se puede ver los distintos obstáculos que éste ve. Finalmente, se puede ver que no hay ninguna perdida sistemática de paquetes a nivel TCP ni de mensajes a nivel de ROS (lo que se vería como sectores circulares vacíos), con lo que se tiene que la cadena de nodos de comunicaciones y visualizador resiste el flujo de datos del radar.

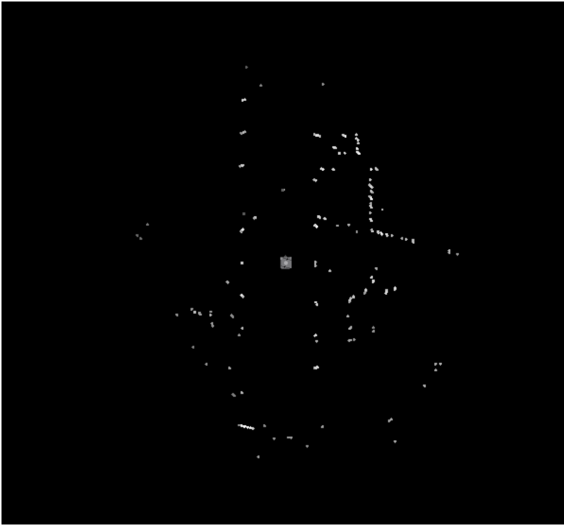


Figura 4.12: Modo de *peak* más potente.

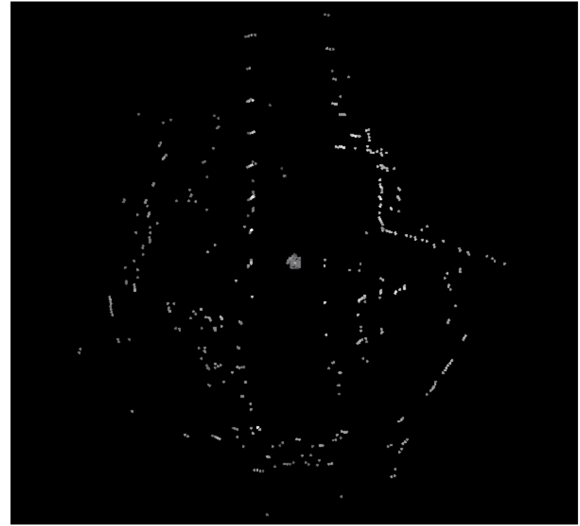


Figura 4.14: Modo de *peak* más lejano.

Figura 4.13: Modo de *peak* más cercano.

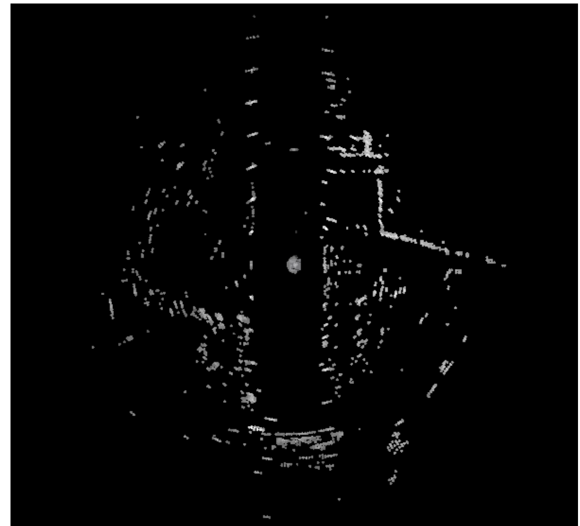


Figura 4.15: Modo de todos los *peaks*.

4.4.2. Detección de obstáculos

Para esta sección se usó una nube de puntos de espectro en frecuencia que se muestra en la Figura 4.16.

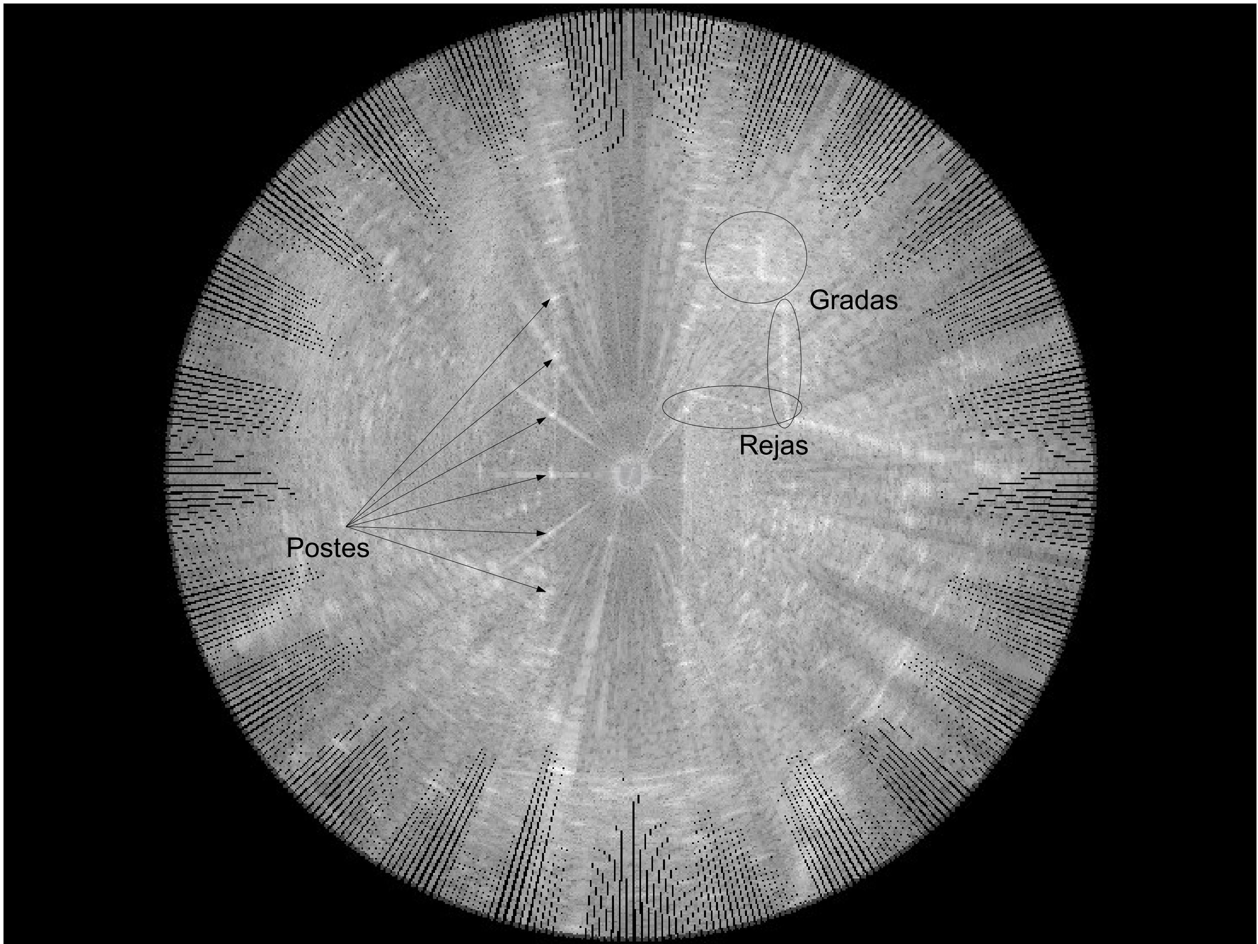


Figura 4.16: Vista de espectro en frecuencia. Las medidas en negro y blanco son de bajas y altas potencias recibidas respectivamente.

Un detalle relevante para los experimentos siguientes es el hecho de que los postes generan líneas completas de altas potencias, de forma que en la Figura 4.16 se ven líneas radiales de altas potencias que pasan por los postes. Estas reflexiones múltiples se dan porque estos obstáculos tienden a reflejar en varias frecuencias, lo que el radar interpreta como varias distancias [5, Sección 2.7.1.4].

Usando los datos obtenidos de espectro en frecuencia se aplicó el procesador CFAR con los siguientes parámetros:

- Probabilidad de falsa alarma: 10^{-5}
- Ancho de ventana de referencia: 3 m
- Ancho de ventana de guardia: 0,5 m

El resultado de aplicar los procesadores CFAR tridimensional y unidimensional se muestran en las Figuras 4.17 y 4.18.

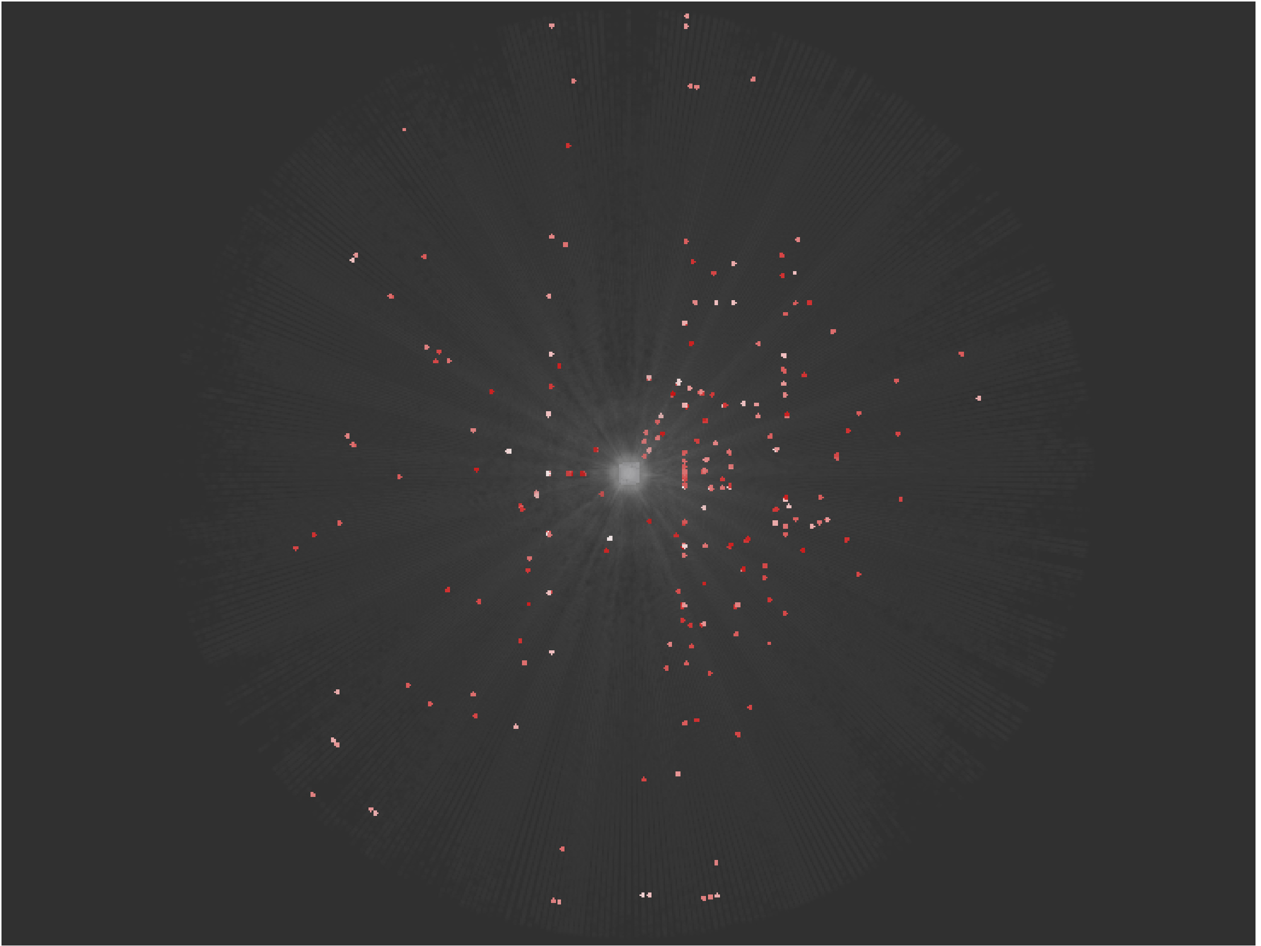


Figura 4.17: Obstáculos detectados por el procesador CFAR tridimensional. Las medidas en rojo y blanco son las detecciones de menor y mayor P_D^{CFAR} respectivamente.

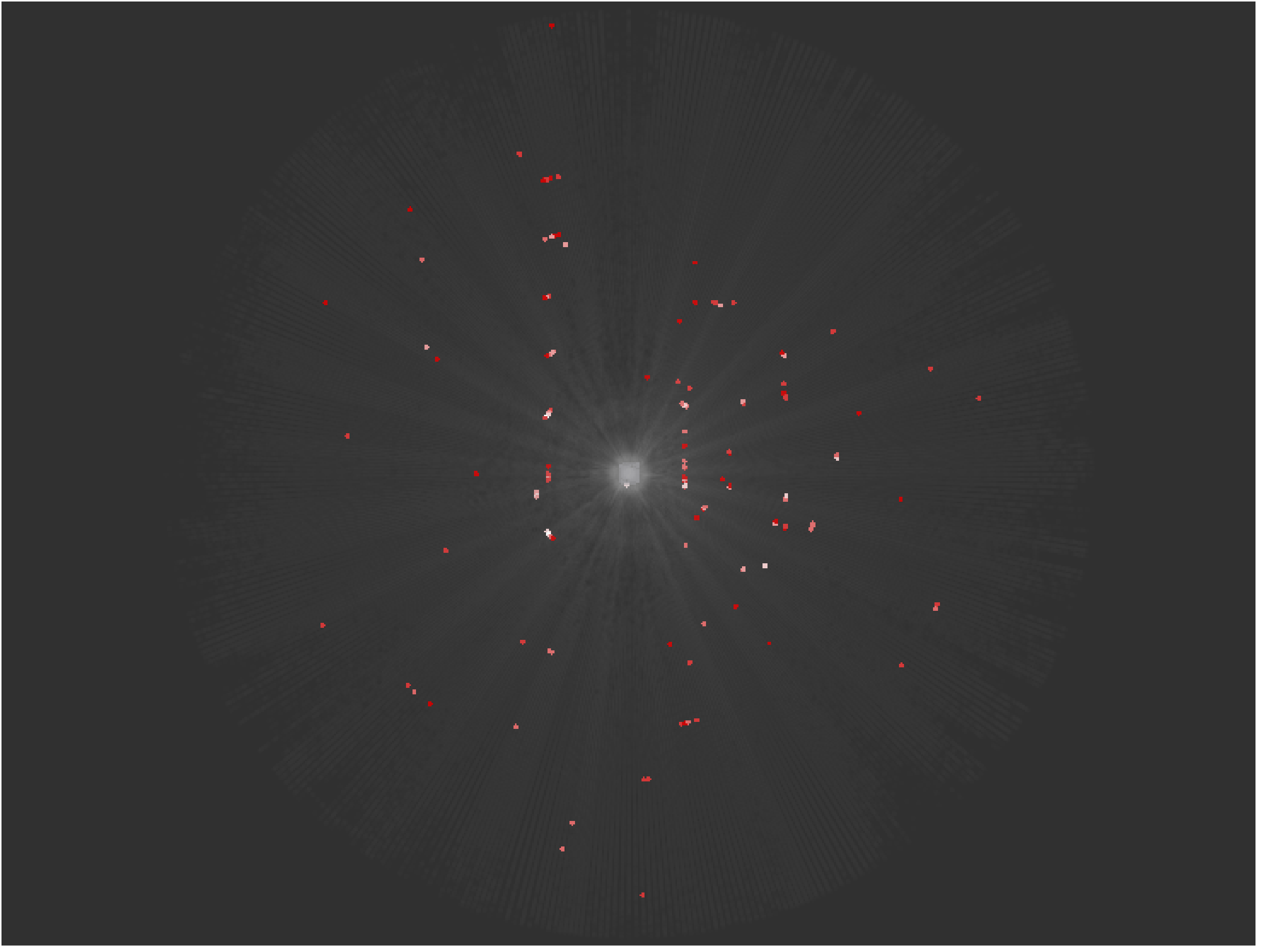


Figura 4.18: Obstáculos detectados por el procesador CFAR unidimensional. Las medidas en rojo y blanco son las detecciones de menor y mayor P_D^{CFAR} respectivamente.

En las Figuras 4.17 y 4.18 se puede ver que el procesador CFAR tridimensional detecta más obstáculos que el unidimensional. Esto se da porque el procesador tridimensional hace diferentes estimaciones del ruido ambiental en un punto, al tomar más datos que el procesador unidimensional. También se puede ver que algunos de los obstáculos detectados, especialmente los postes, en el caso unidimensional, generan varias mediciones agrupadas en líneas, mientras que en el caso tridimensional solo se detecta un punto. Esto es porque en el caso unidimensional no se tiene, para un *A-scope*, información sobre los otros *A-scopes*, de forma que en cada uno se detecta un obstáculo separadamente.

Otro efecto que se da con el procesador CFAR tridimensional es que, al contrario del procesador unidimensional, éste falla en descartar las múltiples reflexiones de los postes. Esto también se da porque el procesador tridimensional toma *A-scopes* separados mientras que el tridimensional toma puntos de distintos *A-scopes*. En el procesador unidimensional, las medidas que se toman para hacer el test estadístico de una medida correspondiente a una reflexión múltiple están en el mismo *A-scope*, y también son reflexiones múltiples, entonces el umbral necesario para que una medida se detecte como ruido es alto, porque para cada

ventana que toma el procesador, todos los puntos son de alta potencia. En cambio, usando el procesador tridimensional, también se toman muestras de otros *A-scopes*, donde no hay reflexiones múltiples y la potencia recibida es baja, con lo que el umbral para la detección se vuelve más bajo, haciendo que sea más fácil que una medida se detecte como obstáculo. No obstante este problema, también se puede ver que las mediciones procesadas entregan la probabilidad de detección de cada medida, y en el caso de las reflexiones múltiples, estas son de baja probabilidad de detección (rojo), mientras que los verdaderos postes son identificados con alta probabilidad de detección (blanco), de forma que los obstáculos detectados con el procesador son distinguibles entre sí.

Un problema encontrado con el procesador CFAR tridimensional fue el tiempo que toma en procesar una nube de puntos es demasiado para una aplicación en tiempo real. Usando una nube de 226304 puntos, la implementación tridimensional toma 12,881 s, mientras que la implementación unidimensional toma 0,33 s, lo cual, considerando una rapidez del *scanner* de 1 Hz, permite el procesamiento de datos en tiempo real.

4.4.3. Comparación de fuentes de datos

Como se ve en la Figura 4.11, el sistema incluye un sensor láser. Se probó la integración de ambos sensores, obteniéndose los resultados de la Figura 4.19.

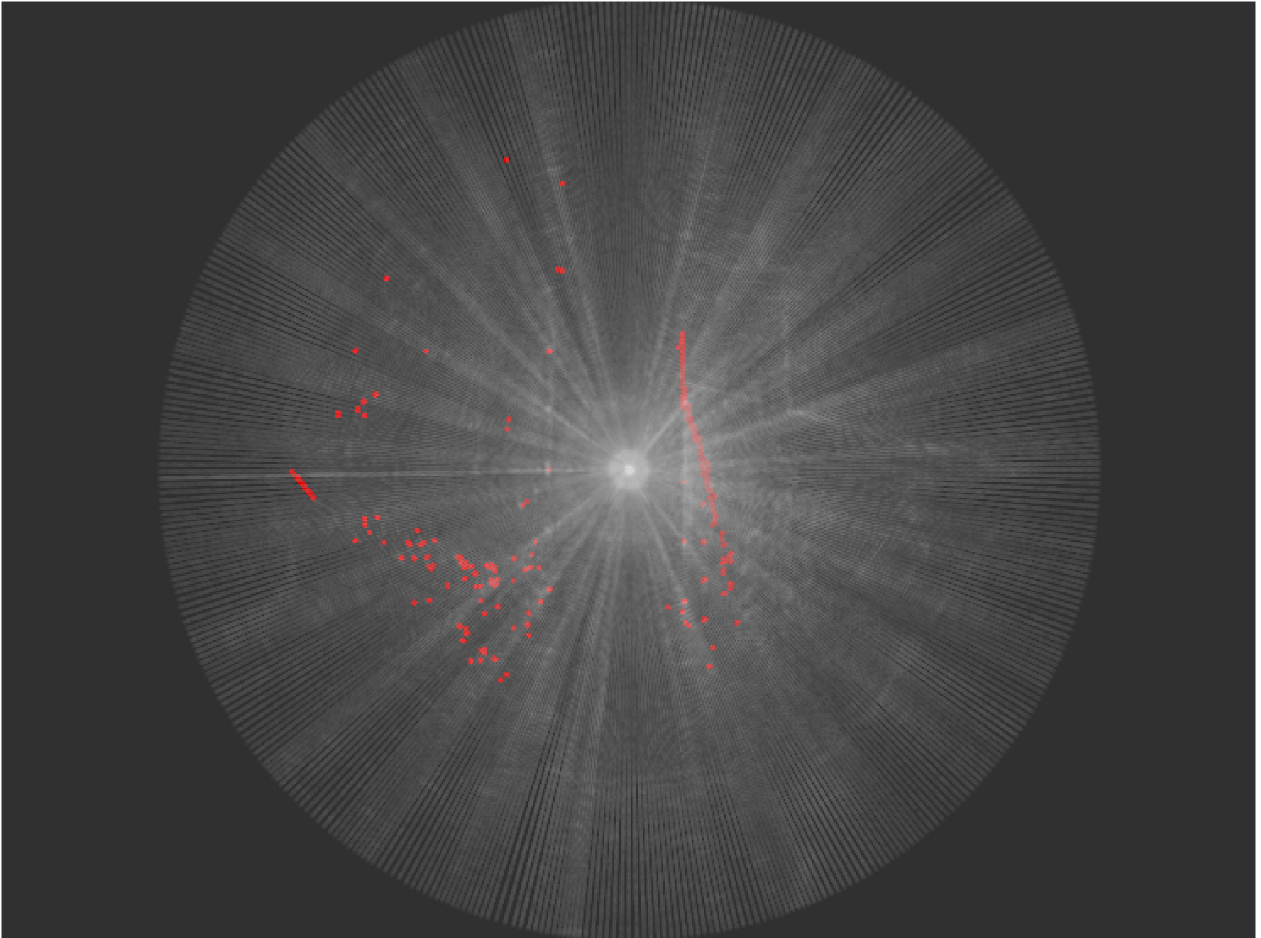


Figura 4.19: Medidas obtenidas por el sensor láser en conjunto con las medidas del radar. Las primeras son de color rojo, mientras que las segundas son en tonos de blanco y negro.

En la Figura 4.19 se ve la principal diferencia entre estos dos sensores. Mientras que el radar obtiene, para un ángulo, un conjunto de medidas de potencia contra distancia (un *A-scope*), el sensor láser solo obtiene un punto por ángulo, y además, en el caso del sensor láser usado para este experimento, éste no entrega información sobre la potencia asociada a un punto, al contrario del radar. Las diferencias entre la información obtenida por ambos sensores se pueden ver, por ejemplo, en el sector central derecho, donde el radar obtiene muchos *peaks* de potencia, y el láser obtiene una línea oblicua. Este sector, como muestra la Figura 4.10, corresponde a pasto, que además está con un ligero desnivel con respecto al plano del robot. Esto genera ruido para el radar, pero para el láser sólo genera mediciones en rangos erróneos. Esto muestra que el láser no puede obtener más información sobre la naturaleza del entorno, es decir, el nivel de ruido ambiental, al contrario del radar. El procesador CFAR, como se ve en la Figura 4.17, puede tomar ventaja de toda la información generada por el radar y obtener un número más realista de obstáculos.

Finalmente, el efecto de las múltiples reflexiones de los postes se puede corregir tomando medidas desde otro punto. Se tomaron mediciones con el radar a aproximadamente 50 m de

las que se muestran en la Figura 4.16, con las que se pudo construir una medición conjunta, que se muestra en la Figura 4.20.

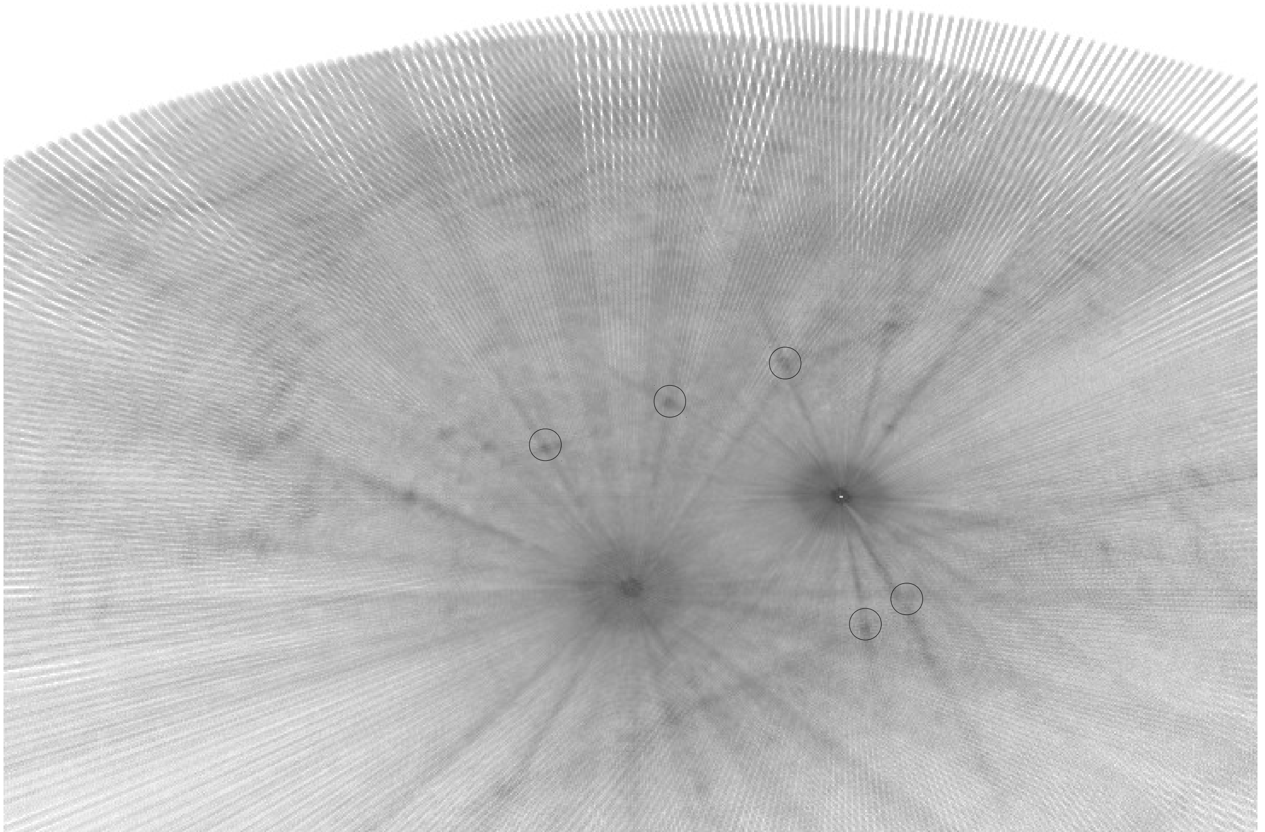


Figura 4.20: Mediciones unificadas entre dos puntos. Los tonos de blanco y negro corresponden a bajas y altas intensidades respectivamente. Los círculos en negro corresponden a cruces entre dos pares de reflexiones múltiples. Comparado con la Figura 4.16, en esta Figura la cámara se ubica a la derecha y mira hacia la izquierda.

En la Figura 4.20 la corrección de la posición de las dos medidas fue hecha manualmente, analizando las coincidencias de los obstáculos representativos, como las rejas. Entonces, de estos datos se pueden tomar los verdaderos obstáculos tomando las intersecciones de pares de reflexiones múltiples.

Capítulo 5

Conclusiones

En esta memoria se implementó una librería de control de un radar con interfaz a la plataforma ROS, y se desarrolló una aplicación de detección de obstáculos usando los datos del radar. Se logró visualizar los datos obtenidos por el radar en tiempo real, y se compararon dos implementaciones de un procesador CFAR.

La librería implementada se compone de tres nodos que permiten la comunicación con el radar, de forma robusta y capaz de manejar el caudal de datos del radar, de forma de poder leer los datos que emite el radar y enviar datos a éste para configurarlo. Con esto se obtuvo un sistema mínimamente funcional, que permite el control del radar mediante línea de comandos y guardar datos en un formato ad-hoc para el radar. Sobre estos nodos se construyó una interfaz gráfica que se comunica con los nodos de comunicaciones mediante llamadas a servicios (una de las formas estándares de comunicación en ROS), un nodo que permite la emisión de nubes de puntos a partir de los datos enviados desde el radar, y un nodo de post-procesamiento de esos datos, usando un procesador CFAR. Al final del proyecto se obtuvo un sistema capaz de aprovechar todas las características del radar y de funcionar en conjunto con otros sensores en la plataforma ROS.

Una de las dificultades encontradas en esta memoria fue que el protocolo [1] tiene algunas fallas con respecto al radar utilizado, de forma que al momento de implementar hubo que hacer una suerte de ingeniería inversa al radar, haciendo que el proceso no fuera tan simple como plasmar el protocolo automáticamente en el código. De todas formas, con los datos del radar y la especificación del protocolo como punto de partida fue posible reconstruir el protocolo de comunicaciones. También el hecho de que ROS sea una plataforma que tiene muchas formas de ser descrita (librería para sistemas distribuidos, plataforma de desarrollo, administrador de paquetes, etc.) hizo que fuese difícil trabajar al comienzo (como suele ser con cualquier otra tecnología desconocida). No obstante, como ROS es una plataforma de desarrollo activa, fue posible encontrar toda la información necesaria para trabajar en los documentos oficiales, o consultando a la comunidad de desarrollo.

Uno de los aportes de la memoria, que hicieron el trabajo más simple, fue que al hacer el

trabajo integrado con ROS se logró utilizar fácilmente las herramientas que esta plataforma ofrece, haciendo que con menos trabajo se obtuvieran más funcionalidades. Un ejemplo es que en [11] se hizo un visualizador de datos desde cero, en cambio en este trabajo se pudo usar Rviz para visualizar los datos.

Finalmente, se hicieron pruebas en terreno, usando el radar en una configuración de uso real. Se vio la integración de los datos del radar con los datos generados por otro sensor, en ese caso un láser, con lo que se puede hacer una fusión de información de sensores de forma transparente, una aplicación básica en robótica [2].

5.1. Trabajos futuros

Entre los desarrollos que pueden mejorar el uso del radar están los siguientes:

- Agregar correcciones por movimiento. Esto es que, al usar el radar con el *scanner* en movimiento, las nubes de puntos son generadas con centro en el radar. Entonces no se pueden tomar datos mientras el radar mismo está en movimiento. La mejora sería que el nodo `clouds` incorpore medidas de desplazamientos, generados por alguna base móvil, como la usada en los experimentos, y use esa información para la generación de las nubes de puntos.

- Analizar y optimizar el detector CFAR tridimensional. Actualmente este algoritmo toma tiempos demasiado largos para su utilización en tiempo real. Hay que analizar el funcionamiento de la librería FLANN para optimizarla, o usar otro método para el cómputo del procesador CFAR tridimensional.

Bibliografía

[1] Martinez, Javier: *Procedures for the Assembly and Operation of the Acumine 94GHz High Speed Scanning Radar System*. 2012.

[2] Adams, Martin: *Apunte de curso Robotics, Sensing and Autononomous Systems*.

[3] *Advanced Mining Technology Center*. `http://www.amtc.cl`, 14 Julio 2014.

[4] *Introduction to ROS*. `http://wiki.ros.org/ROS/Introduction`, 14 Julio 2014.

[5] Adams, Martin, John Mullane, Ebi Jose y Ba Ngu Vo: *Robotic navigation and mapping with radar*. Artech House, 2012, ISBN 978-1-60807-482-2.

[6] *ROS: Technical Overview*. `http://wiki.ros.org/ROS/Technical%20Overview`, 14 Julio 2014.

[7] Gamma, Erich, Richard Helm, Ralph Johnson y John Vlissides: *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995, ISBN 0-201-63361-2.

[8] *Rviz: Herramienta de visualización tridimensional para ROS*. `http://wiki.ros.org/rviz`, 14 Julio 2014.

[9] *Rqt_plot: Herramienta de visualización escalar para ROS*. `http://wiki.ros.org/rqt_plot`, 14 Julio 2014.

[10] Ogata, Katsuhiko: *Modern Control Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001, ISBN 978-0-13-615673-4.

[11] Chin, Tan Hui: *Radar Data Collection for Environmental Mapping*. 2009.

[12] *Python interface to Tcl/Tk*. `https://docs.python.org/2/library/tkinter.html`, 14 Julio 2014.

[13] *WxWidgets bindings for the Python programming language*. `http://wxpython.org`, 14 Julio 2014.

[14] *ROS OpenNI*. `http://wiki.ros.org/openni_camera`, 14 Julio 2014.

[15] Rusu, Radu Bogdan y Steve Cousins: *3D is here: Point Cloud Library (PCL)*. En *IEEE International Conference on Robotics and Automation*, Shanghai, China, Mayo 2011.

Anexos

A. Interfaz a ROS del cliente implementado

En esta sección se muestra la interfaz del radar a ROS. Todos los nombres en esta sección, de tópicos, servicios y parámetros, comienzan con el prefijo `/acumine_radar/`. Los tópicos y servicios se muestran con su nombre, el tipo del mensaje o servicio (definidos en las carpetas `msg` y `srv` respectivamente) y una descripción de éstos.

Tópicos publicados

La definicion de los mensajes publicados se encuentra en la carpeta `msg` del paquete.

- `radar/peak: Sight`
  Medidas del radar en los distintos modos de *peaks*.
- `radar/spectrum: Spectrum`
  Medidas del radar en modo de espectro en frecuencia.
- `radar/analog_signal: AnalogSignal`
  Medidas del radar en modo análogo.
- `scanner/pose: ScannerPose`
  Información de pose del *scanner*.
- `scanning_radar/peak: PeakPose`
  Medidas conjuntas de radar y *scanner* en los distintos modos de *peaks*.
- `scanning_radar/spectrum: SpectrumPose`
  Medidas conjuntas de radar y *scanner* en modo de espectro en frecuencia.
- `clouds: PointCloud2`
  Nubes de puntos con datos del radar girando.
- `processed_clouds: PointCloud2`

Nubes de puntos de datos procesados con el detector CFAR.

Servicios ofrecidos

La definición de los servicios se encuentra en la carpeta `srv` del paquete.

- `radar/set_fft_parameters: SetFFTParameters`
  Define los parámetros de la transformada de Fourier que aplica el radar.
- `radar/set_processing_mode: SetProcessingMode`
  Define el modo de procesamiento del radar.
- `scanner/enable_scanner: EnableScanner`
  Activa o desactiva el *scanner*.
- `scanner/set_scanner_trajectory: SetScannerTrajectory`
  Define la posición o la velocidad del *scanner*.
- `scanner/set_pid_parameters: SetPIDParameter`
  Define las ganancias del controlador PID del *scanner*.

Parámetros

Los parámetros que se definen para el uso del radar están definidos en el archivo `config/acumine_radar.yaml` y es necesario que estén todos definidos antes de iniciar los nodos de la librería.

- `ip` : IP del computador del radar.
- `maximum_range`: Rango máximo del radar.
- `minimum_range`: Rango mínimo del radar.
- `radar/port`: Puerto TCP asociado al `RadarServer`.
- `radar/fft_parámeters`: Parámetros de la transformada de Fourier del radar.
- `radar/processing_mode`: Modo de procesamiento del radar.
- `scanner/port`: Puerto TCP del `ScannerServer`.
- `scanner/pid_parameters`: Parámetros del controlador PID del *scanner*.
- `scanner/trajectory`: Posición y velocidad del *scanner*.

- `scanning_radar/port`: Puerto TCP del `ScanningRadarServer`.

B. Diferencias encontradas entre [1] y el verdadero protocolo del radar

A lo largo del desarrollo de la memoria, se encontraron diferencias entre [1] y el protocolo de comunicación del radar usado en esta memoria. Los siguientes son los mensajes que se diferencian de [1].

- **Estructura general**

    - Cabecera: `0xB255`
    - *Timestamp* en segundos: 8 bytes, entero sin signo
    - Nanosegundos: 4 bytes, entero sin signo
    - Largo de mensaje: 4 bytes, entero sin signo
    - Datos: Largo variable, definido por el campo anterior
    - *Checksum*: 2 bytes, entero sin signo
    - Cola: `0x759F`

- **Mensajes de peaks**
  Este tipo de mensaje es el que se encontró en vez de los definidos como *RadarHighestIntensityMessage*, *RadarLowRangePeakMessage*, *RadarHighRangePeakMessage* y *RadarAllPeaksMessage*, siendo solo el identificador de mensaje el campo que diferencia los cuatro tipos de mensajes.

    - Identificador de mensaje: 1 byte, entero sin signo
    - *Timestamp*: 8 bytes, entero sin signo
    - Rango máximo: 4 bytes, punto flotante
    - Resolución: 4 bytes, punto flotante
    - Numero de *peaks*: 1 byte, entero sin signo
    - *Peaks*:
        - Rango: 4 bytes, punto flotante
        - Intensidad: 4 bytes, punto flotante

- ***ScannerTrajectoryControlMessage***

    - Identificador de mensaje: 1 byte, entero sin signo
    - Número de eje: 1 byte, entero sin signo
    - *TrajectorySetPoint*

- ***TrajectorySetPoint***

    - Identificador de comando: 1 byte, entero sin signo
    - Posicion: 4 bytes, punto flotante

- Rapidez: 4 bytes, punto flotante
- Aceleración: 4 bytes, punto flotante

- ***ScannerEnabledControlMessage***

  - Identificador de mensaje: 1 byte, entero sin signo
  - Número de eje: 1 byte, entero sin signo
  - Activación: 1 byte, entero sin signo

- ***ScannerPIDSettingsMessage***

  - Identificador de mensaje: 1 byte, entero sin signo
  - Número de eje: 1 byte, entero sin signo
  - Ganancia proporcional: 2 bytes, entero sin signo
  - Ganancia derivativa: 2 bytes, entero sin signo
  - Ganancia integral: 2 bytes, entero sin signo
  - Límite de integral: 2 bytes, entero sin signo