Dockerización de ROS para despliegue ágil de algoritmos de exploración

Nievas Martin[1][0000-0003-1791-1727], Paz Claudio Jose[1][0000-0002-8673-0666], and Araguás Roberto Gastón[1][0000-0002-2478-5477]

Centro de Investigación en Informática para la Ingeniería (CIII)
Universidad Tecnológica Nacional, Facultad Regional Córdoba (UTN-FRC)
mnievas@frc.utn.edu.ar

Resumen En este trabajo se presenta la utilización de contenedores para realizar pruebas con diferente algoritmos de exploración. Los mismos fueron implementados en el sistema operativo para robots ROS, y evaluados mediante el simulador Gazebo dentro del mismo contenedor. Se diseñaron simulaciones para utilizar dos algoritmos de SLAM y el algoritmo de exploración de fronteras más cercanas. También se analizan las ventajas de utilizar contenedores a la hora de realizar las pruebas. El presente trabajo fue desarrollado en ROS Kinetic Kame y simulado en Gazebo 7, los cuales se ejecutan mediante un contenedor de Docker. Se presentan los resultados obtenidos, como también un repositorio en el cual puede encontrarse el archivo Dockerfile y los algoritmos utilizados en las pruebas.

Keywords: Exploración · ROS · Simulación · Docker

1. Introducción

La exploración autónoma puede considerarse como el proceso en el cual un robot móvil se desplaza en un ambiente, e incrementalmente construye un mapa del entorno a través de las mediciones de sus sensores. Este mapa del entorno, es útil para poder desplazarse y planificar trayectorias sobre él. Debido a que no siempre se dispone de robots físicos, o el tiempo de preparación de los mismos es grande, en la práctica el primer paso para probar o comparar un algoritmo de exploración es mediante simulación. Dentro de los simuladores 3D, se pueden nombrar: MORSE (Modular Open Robots Simulation Engine) [1], Webots [2], USARSim [3], Gazebo [4] y V-REP [5] entre otros. En los últimos años V-REP y Gazebo han tenido un gran uso en diferentes tareas relacionadas con la exploración utilizando robots [6,7]. Esto se debe, en gran parte, a su integración con ROS (Robot Operating System) [8], el cual a su vez integra una gran cantidad de paquetes para exploración, algoritmos de planificación de rutas y de construcción de mapas. Sin embargo configurar el entorno no es una tarea trivial, ya sea por problemas de compatibilidad, por la inexistencia de ciertos paquetes en una determinada versión, o porque la instalación en el ambiente de trabajo personal requiere de ciertas dependencias que no pueden ser satisfechas.

Una de las opciones para aislar el entorno de trabajo del local es mediante máquinas virtuales. Como puede verse en [9] es posible configurar entornos para el desarrollo con ROS utilizando maquinas virtuales. Otra opción puede ser utilizar contenedores Linux, que permiten empaquetar y aislar las aplicaciones junto con todo el entorno necesario durante su tiempo de ejecución. Sin embargo, en trabajos recientes [10] se muestra que los contenedores Linux producen un rendimiento igual o mejor que las máquinas virtuales en casi todos los casos analizados. Docker [11] es una plataforma de código abierto capaz de ejecutar aplicaciones en forma aislada sobre el kernel del sistema operativo, lo que permite un proceso más fácil de desarrollo y distribución de algoritmos.

En este trabajo se describe el proceso para crear y utilizar los contenedores de diferentes algoritmos de exploración, utilizando como entorno de simulación Gazebo en combinación con ROS, y Docker como plataforma de aislación. El trabajo se organiza de la siguiente manera: en la Sección 3 se realiza una introducción al entorno de ROS y Gazebo utilizado para la simulación. Luego, en la Sección 4 se presenta la descripción del contenedor utilizado para realizar las simulaciones mediante Docker. En la Sección 5 se enumeran los algoritmos utilizados para las diferentes pruebas. Los resultados de los algoritmos utilizados pueden encontrarse en la Sección 6, mientras que en la Sección 7 se analizan los resultados y los posibles trabajos futuros.

2. Estado del arte

En los últimos años la robótica, como área de investigación, ha demostrado un gran interés por la reproducción de resultados que se publican [12,13]. Sin embargo, pese a que se avanza hacia el uso de herramientas de programación que permiten a los investigadores tener una base común de desarrollo, como ROS [14], Gazebo [4] o YARP [15], los sistemas son cada vez más complejos y difíciles de reproducir. Sumado a esto, no siempre se indican todas las herramientas o dependencias utilizadas en el proceso de investigación para lograr su reproducción.

Uno de los primeros trabajos en utilizar Docker en la robótica es [16], en el cual los autores proveen un tutorial de cómo integrar ROS dentro de un contenedor, y enumeran un conjunto de ventajas al utilizar Docker en aplicaciones de robótica. Entre ellas se pueden destacar la repetibilidad y la reproducibilidad de los experimentos de robótica, de gran utilidad tanto en los ámbitos de la educación, la investigación como en la industria. Más recientemente, Cervera [17] explica que la utilización de Docker para el ámbito de la investigación en robótica no interfiere con el flujo normal de trabajo, y permite fácilmente reproducir el entorno utilizado por los autores. También reconoce que al correr el software de forma local, permite la interacción con las interfaces gráficas de usuario, como por ejemplo ROS y RViz. Wang et al. [18] proponen la utilización de un esquema basado en contenedores Linux (LXC) para administrar, programar y monitorear componentes de software en robots de servicio. En [13] Weisz et al. presentan una plataforma (RoboBench) la cual permite compartir simulaciones de sistemas

robóticos y realizar evaluaciones comparativas. RoboBench utiliza contenedores Linux, los cuales son integrados a la plataforma CITK [19]. Esta plataforma permite recrear, construir y desplegar experimentos de robótica, como también la descripción, repetitividad y ejecución.

Estos trabajos anteriormente mencionados, si bien utilizan los contenedores Linux y ROS, no plantean específicamente el problema de la exploración robótica, el cual al ser un enfoque más completo que la construcción de mapas abordada en los ejemplos mencionados en [13], requiere de un tratamiento especial. En este trabajo se presenta además de la configuración del entorno utilizado mediante contenedores Linux, un repositorio que cuenta con todos los algoritmos utilizados, el cual está disponible para ser descargado y editado, y cuenta con la ventaja de estar configurado para el entorno de simulación provisto.

3. Gazebo y ROS

En el presente trabajo se utiliza como simulador a Gazebo 7. El mismo se eligió por tener una gran cantidad de modelos disponibles para la simulación, tanto a nivel de robots disponibles para utilizar, como de elementos físicos para incluir en los ambientes. Por otro lado, cuenta con plugins disponibles para la simulación de sensores, entre los cuales podemos mencionar: sensor de rango láser, unidades de medición inercial (IMU), cámaras RGB y de profundidad entre otros. Con el objeto de disminuir la carga computacional extra en la simulación, se diseñaron dos nuevos robots, mecánicamente muy simples pero con capacidad para utilizar los algoritmos de exploración disponibles en ROS. Estos robots, al ser de configuración diferencial, pueden ser controlados mediante los paquetes de control y navegación incluidos en ROS. Físicamente ambos son idénticos, a diferencia que uno de ellos posee una cámara orientada hacia el frente del robot. La principal ventaja de utilizar Gazebo, es que posee una gran integración con el sistema operativo ROS, el cual permite mediante la utilización de mensajes, enviar y recibir información integrando todos los sensores y robots empleados en los experimentos.

ROS permite controlar cada robot mediante la utilización de nodos conectados entre sí. Estos nodos están dispuestos en una jerarquía, en donde la más baja corresponde al Hardware del robot. El resto del sistema está construido sobre esta última capa jerárquica. En una simulación mediante Gazebo, el robot se trata como un nodo que describe la capa Hardware, ROS luego se comunica con estos nodos de capa simulados. ROS cuenta con una gran cantidad de paquetes para realizar tareas de exploración. Entre los paquetes disponibles se encuentran algoritmos de SLAM (simultaneous localization and mapping) como `gmapping` y `hector slam`. Si bien estos algoritmos están implementados para 2D, otros están dedicados a la representación de mapas 3D, como OctoMap [20]. ROS también cuenta con un algoritmo de exploración llamado `frontier_exploration`, utilizando el concepto de fronteras más cercanas, introducido por Yamauchi [21]. ROS está disponible en diferentes versiones, para el trabajo actual se utilizó `ROS`

`Kinetic Kame`, la cual se eligió ya que cuenta con soporte e implementación estable de todos los módulos utilizados para las pruebas.

4. Docker

Docker es un proyecto de código abierto basado en diferentes tecnologías para la investigación de sistemas operativos [22], entre las cuales podemos destacar los contenedores Linux (LXC) y la virtualización de sistemas operativos. Una de las diferencias entre las máquinas virtuales y las imágenes de Docker, es que éstas últimas comparten el kernel Linux con la máquina host. Esto trae como consecuencia que las imágenes de Docker estén basadas en un sistema Linux y utilicen software compatible con el mismo. Una ventaja de las imágenes de Docker es que, al ser creadas interactivamente, dejan un registro de todos los cambios efectuados. Con esto se pueden conocer exactamente las dependencias utilizadas. Docker incluye un script (Dockerfile) similar a un Makefile, que define como construir la imagen especificando todas sus dependencias. Este script es un texto plano, por lo que es fácil de almacenar y compartir. También resulta muy útil para el control de versiones para mantener un registro de los cambios que se fueron realizando.

5. Algoritmos ROS

Los algoritmos implementados pueden separarse en dos categorías: algoritmos manuales y algoritmos autónomos. En los manuales, el operador debe guiar al robot a través del escenario para ir reconstruyendo una representación del ambiente (mapa). Este guiado manual puede ser mediante el paquete de teleoperación o mediante el `navigation_stack`, provistos por ROS. El `navigation_stack` se utiliza para integrar la información de los sensores y generar una ruta. Toma información de la odometría, los sensores y la posición objetivo para producir comandos de velocidad seguros y enviarlos al robot. La odometría viene a través del mensaje de ROS `nav_msg/Odometry`, el cual provee una estimación de la posición y la velocidad del robot para determinar su ubicación. El `navigation_stack` no necesita contar con un mapa a priori para iniciar. Si no se cuenta con uno, el robot solo detectará los obstáculos que estén más próximos y construirá un mapa con el ambiente visto por sus sensores, utilizando el algoritmo de `gmapping` o `hector_slam` según su configuración. Para el caso que el destino final del robot se encuentre más allá del mapa, el robot se desplazará por el entorno intentando llegar. En caso de aparecer un obstáculo en el camino, primero el algoritmo encargado de generar el mapa actualizará el mismo con el obstáculo, y luego el `navigation_stack` procederá a recalcular la ruta. Para el presente trabajo se programó un paquete que permite la manipulación del robot mediante el teclado. El mismo es una copia del paquete provisto por `turtlebot_teleop`, pero que sólo cuenta con la interfaz de teclado. Esto se hizo para evitar la instalación de paquetes que agregan funciones no necesarias para el presente trabajo. Como algoritmo autónomo, se utilizó el paquete `fontier_exploration` para ROS,

mencionado anteriormente. Tanto para el control autónomo como manual, el mapa es generado utilizando los algoritmos `gmapping` o `hector_slam`. El primero es una implementación del algoritmo provisto por [23]. Utiliza un enfoque basado en un filtro de partículas para construir una grilla de probabilidad de ocupación. Por otro lado, `hector_slam` también genera el mismo tipo de mapa, pero está basado en un filtro de Kalman Extendido. El paquete `move_base` si bien también genera dos mapas: `global_costmap` y `local_costmap`; estos son utilizados solamente para la evasión de obstáculos y para la navegación a través del entorno.

6. Experimentación

Los resultados que se muestran a continuación, fueron realizados dentro del contenedor Docker. El archivo de construcción (Dockerfile) junto a los algoritmos y escenarios utilizados, pueden descargarse de:

`https://gitlab.com/martinnievas/ros-jaiio-2019`

En el mismo pueden encontrarse también, los pasos para correr cada una de las simulaciones.

En la Figura 1 pueden observarse los dos modelos de robots utilizados en las simulaciones. Ambos cuentan con un escáner láser en 2D de 180 grados orientado hacia el frente del robot, y la misma estructura física. La única diferencia entre ellos es que uno posee una cámara ubicada apuntando hacia el frente. Se escogió implementar por separado un modelo con cámara, debido a que la renderización de la vista producida por la cámara incrementa el costo computacional de la simulación. El alcance del sensor láser se limitó a 2,5m, mientras que la velocidad lineal está limitada a 0,5m/s para los algoritmos de exploración. Para el caso de la operación manual, el límite de velocidad puede modificarse desde el teclado.

Se eligieron tres ambientes para realizar las pruebas de los algoritmos de exploración. El primero (2a) corresponde a una habitación de 6m $\times$ 6m, delimitado por cuatro paredes sin aberturas. Luego, (2b) corresponde a un ambiente de oficinas con una puerta y todas conectadas por un pasillo. Por último (2c) es el mismo ambiente de oficinas pero amueblado.

Los pasos a realizar para ejecutar un simulación son:

- Construir la imágen de Docker.
- Ejecutar el contenedor Docker.
- Configurar y elegir el escenario.
- Insertar un robot en la simulación.
- Controlarlo manualmente o en forma autónoma.

Todos estos pasos pueden encontrarse mas detallados dentro del repositorio.

La imagen de Docker fue construida mediante:

```
docker build -t ros:jaiio2019 .
```

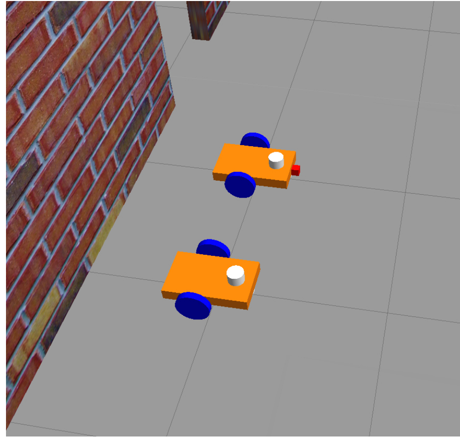
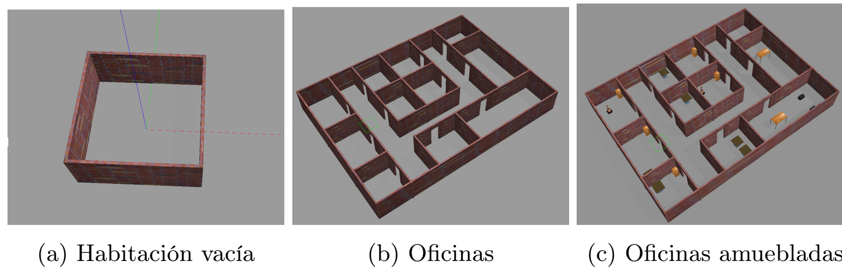


Figura 1: Robots utilizados en la simulación. Puede observarse que el modelo en la parte superior tiene en su frente una cámara, renderizada como un cubo rojo



(a) Habitación vacía (b) Oficinas (c) Oficinas amuebladas

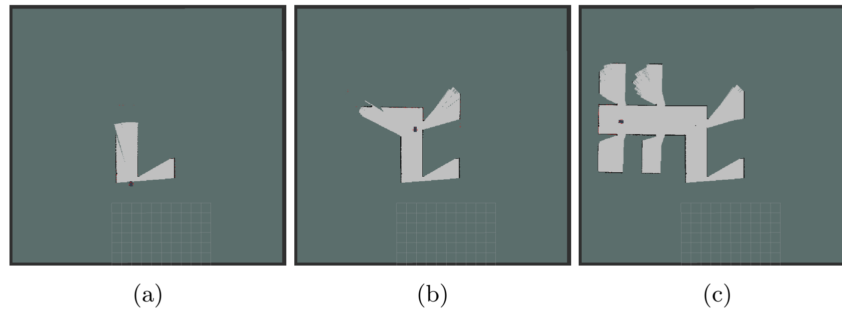
Figura 2: Ambientes a explorar

Donde `ros:jaiio2019` es el nombre de la imagen a crear. Luego, se creó el contenedor con el nombre `ros_jaiio2019` a partir de la imagen anterior, teniendo en cuenta que es necesario pasarle las variables de entorno necesarias para la interfaz gráfica. Ésto fue realizado con `-v /tmp/.X11-unix:/tmp/.X11-unix -e DISPLAY=unix$DISPLAY`

La ejecución del contenedor se realiza con la opción interactiva (-it), ya que es necesario interactuar con el contenedor generado. El siguiente paso es generar una simulación, eligiendo uno de los tres escenarios de la Figura 2. Para el caso de una oficina, la simulación puede ser ejecutada mediante:

```
roslaunch my_worlds office.launch gui:=true
```

El comando `roslaunch` es propio de ROS, y es el encargado de iniciar las configuraciones necesarias para el simulador. La opción `gui` indica si la interfaz gráfica debe ser mostrada o no. Esta opción está disponible a los fines de reducir el cómputo, debido a la renderización del simulador.



(a) (b) (c)

Figura 3: Proceso de exploración utilizando gmapping

Una vez creado el entorno de simulación, se procedió a insertar un robot dentro del mismo mediante el comando

```
roslaunch kalervo_description spawn_kalervo.launch x:=0 y:=0
```

Las variables x e y configuran la posición inicial del robot, en caso de ser omitidas, la posición inicial es el origen.

Para controlar el robot, se pueden utilizar dos métodos: el manual y el autónomo. Para el caso manual, como se describió anteriormente, se implemento un control mediante el teclado, el cual puede ejecutarse mediante:

```
rosrun teleop_robot teleop_robot.py
```

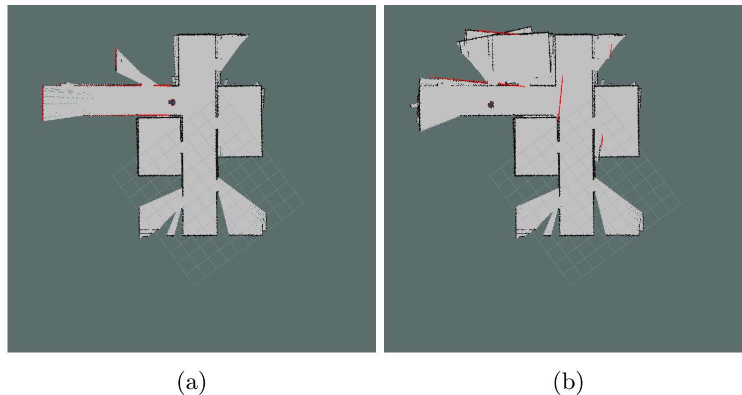
Se decidió separar el algoritmo de mapeo del de control, a los fines de obtener una plataforma modular y experimentar con diferentes configuraciones. Para correr estos algoritmos, en una nueva terminal conectada al contenedor se ejecutó el comando:

```
roslaunch mappig_algorithm kalervo_gmapping.launch
```

El mismo provee las configuraciones para el paquete gmapping implementado en ROS, el cual permite crear un mapa, a medida que el robot es controlado mediante las órdenes de teclado.

En la Figura 3 se observa el proceso de creación del mapa, utilizando el paquete gmapping. El robot fue desplazado de forma manual a través del ambiente. Si bien manualmente se recorrió primero el pasillo, debido al alcance y apertura del sensor láser, parte de las habitaciones son escaneadas durante el recorrido.

Como puede observarse en la Figura 4, el algoritmo para generar el mapa hector_mapping presenta dificultades cuando el robot realiza giros sobre si mismo. Esto es debido a que se producen deslizamientos en las ruedas y al depender de la odometría, el mapa generado presenta inconsistencias. Esto puede solucionarse eligiendo una velocidad más lenta para el desplazamiento del robot, pero aumentando el tiempo total de la exploración. Para ejecutar el algoritmo hector_mapping se utilizaron los comando:



(a) (b)

Figura 4: Proceso de exploración utilizando hector slam

```
roslaunch mappig_algorithm kalervo_hector.launch
```

En la Figura 5 se muestra el proceso por el cual el robot se dirige de forma autónoma hacia un destino previamente indicado, evitando los obstáculos, en este caso, las paredes. El robot a medida que se desplaza construye una representación del ambiente, en este caso utilizando el paquete `gmapping`. Para ésta simulación se utilizaron los comandos:

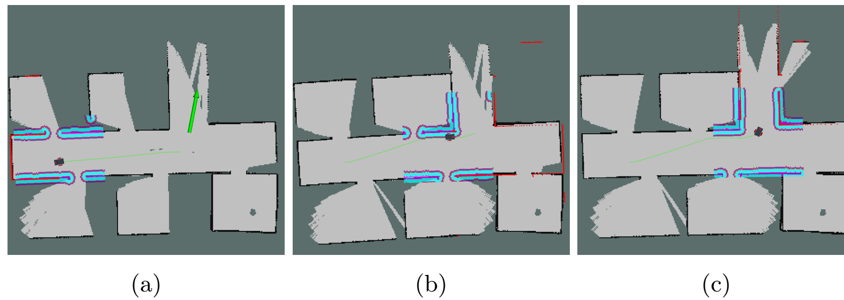
```
roslaunch kalervo_2dnav kalervo_configuration.launch
roslaunch kalervo_2dnav move_base.launch
```

El primer comando realiza las configuraciones para poder controlar el robot mediante el `navigation_stack` provisto por ROS. El segundo comando es el encargado de leer la posición final, y enviarla al `navigation_stack` para generar los controles de velocidad necesarios para alcanzar el destino.

La posición y orientación final son seleccionadas mediante el vector indicado en verde sobre la Figura 5a. En la misma figura puede verse el mapa de costo local (en celeste alrededor de las paredes) utilizado por el algoritmo de planificación para calcular la ruta hacia el destino seleccionado.

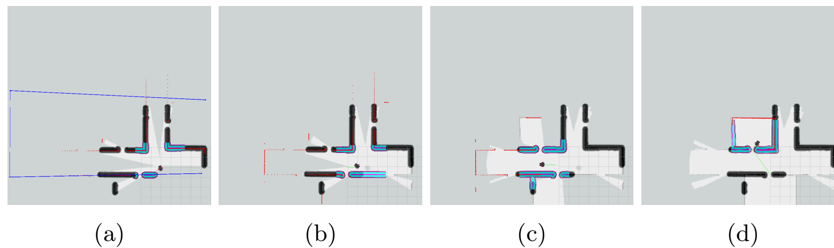
El algoritmo de fronteras más cercanas, necesita un espacio delimitado para determinar el final del proceso de exploración. La misma fue delimitada por el polígono en color azul en la Figura 6a. En la Figura 6 se presenta las primeras etapas del algoritmo de exploración por fronteras más cercanas. Puede observarse como primero recorre el pasillo, y luego, una vez que se completa esa sección, se procede con la habitación superior. Para configurar el paquete `frontier_exploration` es necesario ejecutar los siguientes comandos en orden:

```
roslaunch kalervo_frontier_exploration kalervo_front_explor.launch
roslaunch kalervo_frontier_exploration \
  kalervo_front_explor_algorithm.launch
roslaunch kalervo_frontier_exploration move_base.launch
```



(a) (b) (c)

Figura 5: El robot se dirige hacia el destino seleccionado, y construye progresivamente el mapa, a medida que se desplaza



(a) (b) (c) (d)

Figura 6: El robot explora progresivamente el área delimitada

7. Conclusiones

En este trabajo se utilizó un diseño basado en contenedores, para ensayar diferentes algoritmos utilizados en la exploración robótica. Los mismo fueron implementados en ROS. Se comprobó que mediante la utilización de contenedores Docker se logra aislar el entorno de trabajo del local, lo que reduce las complicaciones en la configuración del entorno. Otra ventaja de correr los contenedores en forma local es que permite la utilización de interfaces gráficas (GUI). Esto último permite visualizar el robot en el simulador Gazebo. Sin embargo, en las últimas pruebas realizadas se comprobó que es posible correr los contenedores en un servidor mediante `ssh` y transmitir la salida de video. Esto tiene como ventaja que reduce la carga en la computadora local, pero incrementa el tráfico de datos en la red debido a la transmisión de video.

Referencias

1. Echeverria, G., Lassabe, N., Degroote, A., Lemaignan, S.: Modular open robots simulation engine: Morse. In: 2011 IEEE International Conference on Robotics and Automation. (May 2011) 46–51
2. Michel, O.: Webots: Symbiosis between virtual and real mobile robots. In: International Conference on Virtual Worlds, Springer (1998) 254–263

3. Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C.: Usarsim: a robot simulator for research and education. In: Proceedings 2007 IEEE International Conference on Robotics and Automation. (April 2007) 1400–1405
4. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566). Volume 3. (Sep. 2004) 2149–2154 vol.3
5. Rohmer, E., Singh, S.P.N., Freese, M.: V-rep: A versatile and scalable robot simulation framework. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems. (Nov 2013) 1321–1326
6. Mohammed, S., Gomaa, W.: Exploration of unknown map for safety purposes using wheeled mobile robots. (01 2017) 359–367
7. Jia, D., Wermelinger, M., Diethelm, R., Krüsi, P., Hutter, M.: Coverage path planning for legged robots in unknown environments. In: 2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR). (Oct 2016) 68–73
8. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. Volume 3., Kobe, Japan (2009) 5
9. Fernandez, E., Crespo, L.S., Mahtani, A., Martinez, A.: Learning ROS for robotics programming. Packt Publishing Ltd (2015)
10. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE international symposium on performance analysis of systems and software (ISPASS), IEEE (2015) 171–172
11. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal **2014**(239) (2014) 2
12. Bonsignorio, F., Del Pobil, A.P.: Toward replicable and measurable robotics research [from the guest editors]. IEEE Robotics & Automation Magazine **22**(3) (2015) 32–35
13. Weisz, J., Huang, Y., Lier, F., Sethumadhavan, S., Allen, P.: Robobench: Towards sustainable robotics system benchmarking. In: 2016 IEEE International Conference on Robotics and Automation (ICRA). (May 2016) 3383–3389
14. Cousins, S., Gerkey, B., Conley, K., Garage, W.: Sharing software with ros [ros topics]. IEEE Robotics & Automation Magazine **17**(2) (2010) 12–14
15. Metta, G., Fitzpatrick, P., Natale, L.: Yarp: yet another robot platform. International Journal of Advanced Robotic Systems **3**(1) (2006) 8
16. White, R., Christensen, H.: Ros and docker. In: Robot Operating System (ROS). Springer (2017) 285–307
17. Cervera, E.: Try to start it! the challenge of reusing code in robotics research. IEEE Robotics and Automation Letters **4**(1) (Jan 2019) 49–56
18. Wang, S., Christensen, H.I.: Tritonbot: First lessons learned from deployment of a long-term autonomy tour guide robot. In: 2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN), IEEE (2018) 158–165
19. Lier, F., Wienke, J., Nordmann, A., Wachsmuth, S., Wrede, S.: The cognitive interaction toolkit–improving reproducibility of robotic systems experiments. In: International Conference on Simulation, Modeling, and Programming for Autonomous Robots, Springer (2014) 400–411
20. Hornung, A., Wurm, K.M., Bennewitz, M., Stachniss, C., Burgard, W.: Octomap: An efficient probabilistic 3d mapping framework based on octrees. Autonomous robots **34**(3) (2013) 189–206

21. Yamauchi, B.: A frontier-based approach for autonomous exploration. In: Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on, IEEE (1997) 146–151
22. Matthias, K., Kane, S.P.: Docker: Up & Running: Shipping Reliable Containers in Production. .O'Reilly Media, Inc."(2015)
23. : OpenSLAM. https://openslam.org/ Accessed: 2019-03-29.